Analysis Techniques for Concurrent Programming Languages

PhD THESIS

Salvador Tamarit Muñoz
Departamento de Sistemas de Información y Computación
Universitat Politècnica de València

Advisors:
Germán Vidal Oriola
Josep Silva Galiana

January, 2013

Contents

Contents

List of Figures

List of figures

Chapter 1

Introduction: Motivation and Contributions

1.1 Concurrent Programming Languages

Nowadays, few computers are based on a single processor architecture. Contrarily, modern architectures are based on multiprocessor systems such as the dual-core or the quad-core; and a challenge of manufacturer companies is to increase the number of processors integrated into the same motherboard. In order to take advantage of these new hardware systems, software must be prepared to work with parallel and heterogeneous components that work concurrently. This is also a necessity of the widely generalized distributed systems, and it is the reason why the industry invests millions of dollars in the research and development of concurrent languages that can produce efficient programs for these systems, and that can be automatically verified thanks to the development of modern techniques for the analysis and verification of such languages.

Concurrency refers to two or more tasks of a computer system which are in progress simultaneously and communicate with each other occasionally. It has been studied since the first concurrency issues appeared in the Operating Systems in the early 1960s. One of the first proposals was the Petri Nets [73]. In the years since, a wide variety of approaches has been announced and discussed, and the number is still being increased nowadays mainly due to the increase of multi-core processors and the Internet model of communication. Notable examples of these approaches are the Parallel Random Access

Machine [31], the Actor model [1] or process calculi [64] such as Ambient calculus [18], Calculus of Communicating Systems (CCS) [64], Communicating Sequential Processes (CSP) [39, 81] or π-calculus [65].

Concurrent programming puts all these ideas in practice. This computing paradigm has many advantages over their peers. One of the most important is to allow various tasks to run in parallel in order to improve the performance. Additionally, this paradigm is the most suitable for the representation of some problems. Trying to define such problems in another paradigm could be a difficult task and result in error prone solutions. Basically, it is a form of computing in which programs are designed as processes which are able to communicate with other processes. Explicit communication between processes fall in one of the following cases:

Shared memory communication All the processes have access to a common shared memory, and the communication is made through it. Commonly, this kind of communication requires to use lockers (e.g. mutexes, semaphores, or monitors) to coordinate all the processes. Java or C# are maybe the most known languages using this class of communication.

Message passing communication The communication is made by exchanging messages. The exchange could be synchronous (when the sender does not progress until the receiver has received the message) or asynchronous (when the sender delivers a message to the receiver and continues its execution without waiting for the receiver to get the message). This kind of communication is usually more robust and easier to model. Most mathematical theories in concurrency allows to understand and analyze this class of communication, including the Actor model [1], and several process calculi [39, 81, 65]. Notable examples of languages using this class of communication are Scala [70] and Erlang [5].

The specification and simulation of complex concurrent systems is a difficult task due to the intricate combinations of message passing and synchronizations that can occur between the components of the system. In particular, the non-deterministic execution order of processes and the restrictions imposed on this order by synchronizations makes the comprehension and development of such systems a costly task. For these reasons, different formalisms exist that are specially useful to solve these problems.

In this thesis we focus on two of the most important concurrent formalisms: the Communicating Sequential Processes (CSP) [39, 81] and the Petri nets [66, 72]. Both formalisms are extensively used to specify, verify and simulate distributed systems.

Communicating Sequential Processes

Process algebras such as *Communicating Sequential Processes* (CSP) [39, 81], π-calculus [63] or LOTOS [8] and process modeling languages such as Promela [40, 68] allow us to specify complex systems with multiple interacting processes. One of the most widespread concurrent specification languages is CSP [39, 81] whose operational semantics allows the combination of parallel, non-deterministic and non-terminating processes. All these features make CSP a powerful language for specifying and verifying concurrent systems. CSP is an expressive process algebra with a big collection of software tools for the specification and verification of complex systems. In fact, CSP is currently one of the most extended concurrent specification languages and it is being successfully used in several industrial projects [15, 34].

Petri Nets

Another extended model is the formalism of Petri nets [66, 72], that are very useful for simulation because they allow us to graphically animate specifications step by step. A Petri net is a graphic, mathematical tool used to model and verify the behavior of systems that are concurrent, asynchronous, distributed, parallel, non-deterministic and/or stochastic. As a graphic tool, they provide a visual understanding of the system; as a mathematical tool, they facilitate its formal analysis.

1.2 Program Analysis of Concurrent Languages

Program analysis of concurrent languages is a complex task due to the non-deterministic execution order of processes. If the concurrent language being studied allows process synchronization, then the analyses are even more com-

plex (and thus expensive), e.g., due to the phenomenon of *deadlock*. Many analyses such as deadlock analysis [47], reliability analysis [44], and refinement checking [80] try to predict properties of the specification which can guarantee the quality of the final system.

Many of these analyses for concurrent languages have been successfully applied in different industrial projects. However, the cost of the analyses performed is usually very high, and sometimes prohibitive, due to the complexity imposed by the non-deterministic execution order of processes and to the restrictions imposed on this order by synchronizations.

State space methods are the most popular approach to automatic verification of concurrent systems. In their basic form, these methods explore the transition system associated with the concurrent system. The transition system is a graph, known as the *reachability graph*, that represents the system's reachable states as nodes: there is an arc from one state s to another s', whenever the system can evolve from s to s'. In the worst case, state space methods have to explore all the nodes and transitions in the transition system. This makes the method useless in practice, even though it is simple in concept, due to the state-explosion problem that occurs when it is applied to non-trivial real problems. The technique is costly even in bounded contexts with a finite number of states since, in the worst case, the reachable states are multiplied beyond any primitive recursive function. For this reason, various approaches have been proposed to minimize the number of system states to be studied in a reachability graph [77].

Program Slicing

Program slicing is a method for decomposing programs by analyzing their data and control flow in order to extract parts of them—called program *slices*—which are of interest. This technique was first defined by Mark Weiser [88] in the context of program debugging of imperative programs. In particular, Weiser's proposal was aimed at using program slicing for isolating the program staments that may contain a bug, so that finding this bug becomes simpler for the programmer. In general, program slicing extracts those statements that are (potentially) determining the values computed at some program point and/or variable of interest, referred to as *slicing criterion*. This technique is also useful as a program comprehension technique [82].

Let us illustrate this technique with an example taken from [86]. Fig-

```
(1)  read(n) ;
(2)  i := 1 ;
(3)  sum := 0 ;
(4)  product := 1 ;
(5)  while i <= n do
        begin
(6)        sum := sum + i ;
(7)        product := product * i ;
(8)        i := i + 1 ;
        end ;
(9)  write (sum) ;
(10) write (product) ;
```

(a) Example program.

```
read(n) ;
i := 1 ;

product := 1 ;
while i <= n do
   begin

      product := product * i ;
      i := i + 1 ;
   end ;

write (product) ;
```

(b) Program slice w.r.t. (10,product).

Figure 1.1: Sub-figures 1.1(a) and 1.1(b) show an example of program slicing.

ure 1.1(a) shows a simple program that requests a positive integer number n and computes the sum and the product of the first n positive integer numbers. Figure 1.1(b) shows a slice of this program w.r.t. the slicing criterion (10,product), i.e., variable product in line 10. As it can be seen in the figure, all the computations that do not contribute to the final value of the variable product have been removed from the slice.

Program slices are usually computed from a *Program Dependence Graph* (PDG) [30] that makes explicit both the data and control dependences for each operation in a program. The work by Weiser has inspired a lot of different approaches to compute slices which include generalizations and concretizations of the initial approach. In general, all of them are classified into two classes: *static* and *dynamic*. A slice is said to be *static* if the input of the program is unknown (this is the case of Weiser's approach). On the other hand, it is said to be *dynamic* if a particular input for the program is provided, i.e., a particular computation is considered. A survey on program slicing can be found, e.g., in [84].

Tracing

One of the most important techniques for program understanding and debugging is tracing [23]. A trace gives the user access to otherwise invisible information about a computation. In the context of concurrent languages, computations are particularly complex due to the non-deterministic execu-

tion order of processes and to the restrictions imposed on this order by synchronizations; and thus, a tracer is a powerful tool to explore, understand and debug concurrent computations.

Translation

Generally known as semantics encoding. The most common form of translation is the compilation of source program into byte-code [2]. However, sometimes is needed or useful to translate between different programming languages or formalisms. There are several successful examples of translation used nowadays [43, 9, 22, 3]. A translation needs to preserve a number of properties such as: composition, reductions, termination, equivalence and, in the context of concurrent languages, distribution [87].

There are some notable examples of translations for concurrent programming languages[27, 32, 71, 28, 83, 10]. Their similar features make feasible these translations. Once translation has been defined, a programmer can have the advantages of both languages.

1.3 Main Goals and Contributions

The main goal of this thesis is to introduce different analyses techniques for concurrent languages. Concretely, the analyses presented are: (1) a static analisis technique based on program slicing for explicitly synchronized languages in general, and CSP in particular; (2) theoretical basis for tracking concurrent and explicitly synchronized computations in process algebras such as CSP; (3) a new technique that allows us to automatically transform a CSP specification into an equivalent Petri Net; and (4) two dynamic slicing techniques for Petri Nets. In particular the main contributions of this thesis are the following:

1. **Static Slicing of Explicitly Synchronized Languages**

   We summarize the contributions in the following list:

   (a) We define two new static analyses for process algebras and propose algorithms for their implementation. Despite their clear usefulness

we have not found similar static analyses in the literature. This work is explained in detail in Paper 1 (see Appendix A).

(b) We define the *context-sensitive synchronized control flow graph* and show its advantages over its predecessors. This is a new data structure able to represent all computations of a specification taking into account the context of process calls; and it is particularly interesting for slicing languages with explicit synchronization. A detailed description of this data structure is included in Paper 1. An algorithm to construct it, based on an instrumentation of the operational semantics of CSP, is presented in Paper 2.

(c) We have implemented our technique and integrated it in ProB [49, 16, 50]. We present the implementation and the results obtained with several benchmarks. All information about the implementation can be found in Paper 1.

2. **Tracking CSP Computations**

The main contributions in the subject are:

(a) The formal definition of tracks.

(b) The definition of the first tracking semantics for CSP and the proof that the trace of a computation can be extracted from the track of this computation.

(c) Concretely, we instrument the standard operational semantics of CSP in such a way that the execution of the semantics produces as a side-effect the track of the computation. It should be clear that the track of an infinite computation is also infinite. However, we design the semantics in such a way that the track is produced incrementally step by step. Therefore, if the execution is stopped (e.g., by the user because it is non-terminating or because a limit in the size of the track was specified), then the semantics produces the track of the computation performed so far. This semantics can serve as a theoretical foundation for tracking CSP computations because it formally relates the computations of the standard semantics with the tracks of these computations.

(d) The implementation of the first CSP tracker.

All this work led to Paper 3.

3. **From CSP Specfications to Petri Nets**

We define a fully automatic transformation that allows us to transform a CSP specification into an equivalent Petri net (i.e., the sequences of observable events produced are exactly the same). This result is very interesting because it allows CSP developers not only to graphically animate their specifications through the use of the equivalent Petri nets, but it also allows them to use all the tools and analysis techniques developed for Petri nets. Concretely, the contributions are:

(a) An instrumentation of the standard CSP operational semantics that produces as a side-effect a Petri net equivalente to the computations performed with the semantics.

(b) Simplification algorithms that significantly reduce the size of the Petri nets generated while keeping the equivalence properties.

(c) An implementation of a fully automatic translator from CSP specifications to Petri nets that has been made public (both the source code and an online version).

(d) Proofs of technical results. They prove the termination of the transformation algorithm; and the equivalence between the produced Petri net and the original CSP specification.

This work is presented in Paper 4.

4. **Dynamic Slicing Techniques for Petri Nets**

We explore two different alternatives for the dynamic slicing of Petri nets.

(a) Firstly, we present a slicing technique that extends the Rakow's algorithm [74, 75] considering an initial marking. We show that this information can be useful when analyzing Petri nets and, moreover, it allows us to significantly reduce the size of the computed slice. Furthermore, we show that our algorithm is, in the worst case, as precise as Rakow's algorithm. The cost is bounded by the number of transitions in the Petri net. Therefore, it can be considered a lightweight approach.

(b) Then, we present a second approach that further reduces the size of the computed slice by only considering a particular execution—here, a sequence of transition firings. Clearly, in this case the computed slice is only useful to analyze the considered firing sequence.

Complete information of this work can be found in Paper 5.

1.4 Organization of the Thesis

The thesis has been organized in two parts. The first part is an introduction describing its main contributions without detailed descriptions. All technical details are in the second part that includes all the papers which led to this thesis.

In Chapter 2 we give an overview of the syntax and semantics of a process algebra (CSP) and the formal basis of Petri Nets (PN).

Chapter 3 begins with a short description of the analyses using an intuitive example. In Section 3.1 we show that previous data structures used in program slicing are inaccurate or inappropriate in our context, and we introduce the *Context-sensitive Synchronized Control Flow Graph* (CSCFG) as a solution and discuss its advantages over its predecessors. A description of the algorithm used to produce this data structure is also included in this section. Our slicing technique is presented in Section 3.2 where we introduce two algorithms to slice CSP specifications from their CSCFGs. In Section 3.3 we present our implementation, describe its architecture, and show the results of some experiments that illustrate the efficiency and performance of the tool. Finally, we discuss some related work in Section 3.4.

Chaper 4 describes the work done for tracking CSP specificatons. The chapter starts with a brief introduction. Then, in Section 4.1 we define the concept of track for CSP. In Section 4.2, we explain how we instrument the CSP operational semantics in such a way that its execution produces as a side-effect the track associated with the computation performed. We describe the implementation of the idea in Section 4.3. Finally, Section 4.4 presents and discusses the related work.

The translation from CSP to Petri nets is introduced in Chapter 5. The chapter starts with an introduction that uses an example to present the main ideas. Then, in Section 5.1 the concept of equivalence used in this work is introduced. Section 5.2 presents an algorithm able to generate a Petri net which is equivalent to a given CSP specification. To obtain the Petri net, the algorithm uses an instrumentation of the standard operational semantics of CSP which is also introduced in this section. Then, in Section 5.3 we introduce some algorithms to further transform the generated Petri nets. The transformation simplifies the final Petri net producing a reduced version that is still equivalent to the original CSP specification. In Section 5.4, we describe the *CSP2PN* tool, our implementation of the proposed technique.

Finally, Section 5.5 overviews related work and previous approaches to the transformation of CSP into Petri nets.

The last work is presented in Chapter 6 and describes our dynamic slicing techniques for Petri nets. Section 6.1 describes our first technique to slice Petri nets, while the second technique is introduced in Section 6.2. The chapter concludes with Section 6.3 that discusses some related work.

Finally, Chapter 7 concludes and presents some ideas for future work.

Chapter 2

Preliminaries

In order to make the thesis self-contained, we recall in this chapter the syntax and semantics of CSP, as well as the formal basis for Petri nets.

2.1 The syntax and the semantics of CSP

Figure 2.1 summarizes the syntax constructs used in CSP specifications. A *specification* is viewed as a finite set of process definitions. The left-hand side of each definition is the name of a process, which is defined in the right-hand side (abbrev. *rhs*) by means of an expression that can be a call to another process or a combination of the following operators:

(Prefixing) It specifies that the compound object CO must happen before process P. Compound objects represent events and communications.

(Internal choice) One of the two processes P or Q is chosen non-deterministically.

(External choice) It is identical to internal choice but the choice comes from the external environment (e.g., the user).

(Conditional choice) It is a choice that depends on a condition, i.e., it is equivalent to `if` $Bool$ `then` P `else` Q.

(Sequential composition) It specifies a sequence of two processes. If the first one (successfully) finishes, the second can start.

		Domains		
		$M, N \ldots \in \mathcal{N}$	(Process names)	
		$P, Q \ldots \in \mathcal{P}$	(Processes)	
		$a, b \ldots \in \Sigma$	(Events)	
		$x, y \ldots \in \Sigma_{\mathcal{V}}$	(Events with variables)	
S	::=	$\{D_1, \ldots, D_n\}$	(Entire specification)	
D	::=	$N = P$	(Process definition)	
	|	$N(\overline{EV_n}) = P$	(Parameterized process)	$\overline{EV_n} = EV_1, \ldots, EV_n$
P	::=	M	(Process call)	
	|	$M(\overline{EV_n})$	(Parameterized process call)	
	|	$CO \rightarrow P$	(Prefixing)	
	|	$P \sqcap Q$	(Internal choice)	
	|	$P \,\Box\, Q$	(External choice)	
	|	$P \trianglelefteq Bool \trianglerighteq Q$	(Conditional choice)	
	|	$P \,;\, Q$	(Sequential composition)	
	|	$P \underset{\{\overline{EV_n}\}}{\|\|} Q$	(Synchronized parallelism)	
	|	$P \backslash \{\overline{EV_n}\}$	(Hiding)	
	|	$P[\![\Re]\!]$	(Renaming)	$\Re : \Sigma_{\mathcal{V}} \rightarrow \Sigma_{\mathcal{V}}$
	|	$SKIP$	(Skip)	
	|	$STOP$	(Stop)	
CO	::=	$EV \mid CO?EV \mid CO!EV$	(Compound Object)	
EV	::=	$a \mid v \mid v : T \mid EV.EV$	(Event with Variables)	$T \subseteq \Sigma$, v is a variable
$Bool$	::=	$true \mid false \mid Bool \vee Bool$	(Boolean expression)	
	|	$Bool \wedge Bool \mid \neg Bool$		
	|	$EV_1 = EV_2 \mid EV_1 \neq EV_2$		

Figure 2.1: Syntax of CSP specifications

(Synchronized parallelism) Both processes are executed in parallel with a set $\{\overline{EV_n}\}$ of synchronized events. In absence of synchronizations both processes can execute in any order. Whenever a synchronized event $a \in \{\overline{EV_n}\}$ happens in one of the processes it must also happen in the other at the same time. Whenever the set of synchronized events is not specified, it is assumed that processes are synchronized in all common events. A particular case of parallel execution is *interleaving* where no synchronizations exist (i.e., $\{\overline{EV_n}\} = \varnothing$).

(Hiding) Process P is executed with a set of hidden events $\{\overline{EV_n}\}$. Hidden events are not observable from outside the process, and thus, they cannot synchronize with other processes.

(Renaming) Process P is executed with a set of renamed events specified with the total mapping $\Re$. An event a renamed as b behaves internally as a but it is observable as b from outside the process.

(Skip) It successfully finishes the current process. It allows us to continue the next sequential process if any.

(Stop) Synonymous with deadlock. It finishes the current process and it will not allow the next sequential process to continue if any.

The domain Σ of events contains basic symbols such as a that can be compounded to produce communications:

(Input) It is used to receive a message from another process. For instance, if $A \subseteq \Sigma$ is any set of events and, for each $x \in A$, we have defined a process $P(x)$, then $c?x : A \rightarrow P(x)$ defines the process which accepts any element a of A and then behaves like the appropriate $P(a)$.

(Output) It is complementary to the input. In this case, $c!x$ is used to send message x.

We allow events that have been constructed out of any finite number of parts using the infix dot '.' (which is assumed to be associative), e.g., $c.a$.

we use labels (that we call *specification positions*) to uniquely identify each literal in a specification which roughly corresponds to nodes in the CSP specification's abstract syntax tree. We define a function $\mathcal{P}os$ to obtain the specification position of an element of a CSP specification and it is defined over nodes of the abstract syntax tree of this CSP specification. Formally,

Definition 2.1.1. (Specification position) A *specification position* is a pair (N, w) where $N \in \mathcal{N}$ and w is a sequence of natural numbers (we use Λ to denote the empty sequence). We let $\mathcal{P}os(t)$ denote the specification position of a term t. Each (parameterized) process definition $N = P$ or $N(\overline{x_n}) = P$ of a CSP specification is labeled with specification positions. The specification position of its left-hand side is respectively $\mathcal{P}os(N) = (N, 0)$ or $\mathcal{P}os(N(\overline{x_n})) = (N(\overline{x_n}), 0)$.
The right-hand side is labeled with the call $\texttt{AddSpPos}(P, (N, \Lambda))$; where function `AddSpPos` is defined as follows:

$\texttt{AddSpPos}(P, (N, w)) =$

$$\begin{cases} P_{(N,w)} & \text{if } P \in \mathcal{N} \\ P_{(N,w)}(\overline{x_n}) & \text{if } P \in \mathcal{N} \wedge \overline{x_n} \in \Sigma_{\mathcal{V}} \\ STOP_{(N,w)} & \text{if } P = STOP \\ SKIP_{(N,w)} & \text{if } P = SKIP \\ co_{(N,w.1)} \rightarrow_{(N,w)} \texttt{AddSpPos}(Q, (N, w.2)) & \text{if } P = co \rightarrow Q \\ \texttt{AddSpPos}(Q, (N, w.1)) \backslash_{(N,w)} B & \text{if } P = Q \backslash B \\ \texttt{AddSpPos}(Q, (N, w.1)) [\![\Re]\!]_{(N,w)} & \text{if } P = Q [\![\Re]\!] \\ \texttt{AddSpPos}(Q, (N, w.1)) \ op_{(N,w)} \ \texttt{AddSpPos}(R, (N, w.2)) & \text{if } P = Q \ op \ R \\ & \forall\ op \in \{\sqcap, \Box, \lhd\rhd, ||, ;\} \end{cases}$$

In the following, specification positions will be represented with greek letters $(\alpha, \beta, \ldots)$ and we will often use indistinguishably a term and its corresponding specification position when it is clear from the context.

We now recall the standard operational semantics of CSP as defined by A.W. Roscoe [81]. It is presented in Fig. 2.2 as a logical inference system. A *state* of the semantics is a process to be evaluated called the *control*. The inference system starts with an initial state, and the rules of the semantics are used to infer how this state evolves. When no rules can be applied to the current state, the computation finishes. The rules of the semantics change the states of the computation due to the occurrence of events. The set of possible events is $\Sigma \cup \{\tau, \checkmark\}$. Events in $\Sigma = \{a, b, \ldots\}$ are visible from the external environment, and can only happen with its co-operation (e.g., actions of the user). The special event τ cannot be observed from outside the system and it is an internal event that happens automatically as defined by the semantics.

✓ is a special event representing the successful termination of a process. We use the special symbol Ω to denote any process that successfully terminated.

In order to perform computations, we construct an initial state (e.g., `MAIN`) and (non-deterministically) apply the rules of Fig. 2.2. The intuitive meaning of each rule is the following:

((Parameterized) Process Call) In a process call, the call is unfolded and the right-hand side of process N becomes the new control. In a parameterized process call, the behavior is the same, but in this case we use function *subs* to substitute in $rhs(N)$ all variables $\overline{x_n}$ by the actual values of the parameters $\overline{a_n}$.

(Prefixing) When event *co* occurs, process P becomes the new control. The only way communications are introduced into the operational semantics is via the prefixing operation $co \rightarrow P$. In general, *co* may be a compound object, perhaps involving much computation to work out what it represents. The prefix *co* may represent a range of possible communications and bind one or more identifiers in P. $comms(co)$ is the set of communications described by *co*. We deal only with closed terms: processes with no free identifiers. Using this, it is possible to handle most of the situations that can arise, making sure that each identifier has been substituted by a concrete value by the time we need to know it. For $a \in comms(co)$, $subs(a, co, P)$ is the result of substituting the appropriate part of a for each identifier in P bound by *co*. This equals P if there are no identifiers bound.

(SKIP) After `SKIP`, the only possible event is ✓, which denotes the successful termination of the (sub)computation with the special symbol Ω. There is no rule for Ω (neither for `STOP`), hence, this (sub)computation has finished.

(Internal Choice 1 and 2) The system, with the occurrence of the internal event τ, (non-deterministically) selects one of the two processes P or Q which is the new control.

(External Choice 1, 2, 3 and 4) The occurrence of τ develops one of the branches. The occurrence of an event $e \neq \tau$ is used to select one of the two processes P or Q and the control changes according to the event.

(Conditional Choice 1 and 2) The condition *Bool* is evaluated. If it is *true*, process P is put in the control, if it is $false$, process Q is.

(Sequential Composition 1) In $P;Q$, P can evolve to P' with any event except $\checkmark$. Hence, the control becomes $P';Q$.

(Sequential Composition 2) When P successfully finishes (with event $\checkmark$), Q can start. Note that $\checkmark$ is hidden from outside the whole process becoming τ.

(Synchronized Parallelism 1 and 2) When an event $e \notin X$ or events τ or $\checkmark$ occur in a branch, the corresponding process (either P or Q) evolves accordingly. Note that $\checkmark$ is hidden from outside the whole process becoming τ.

(Synchronized Parallelism 3) When a visible event $a \in X$ happens, it is required that both processes synchronize, P and Q are executed at the same time and the control becomes $P' \underset{X}{||} Q'$.

(Synchronized Parallelism 4) When both processes have successfully terminated the control becomes Ω and the event $\checkmark$ is visible from outside.

(Hiding 1 and Hiding 2) When event $a \in B$ ($B \subseteq \Sigma$) occurs in P, it is hidden, and thus changed to τ so that it is not observable from outside P. Contrarily, when event $a \notin B$ occurs in P, it behaves normally.

(Hiding 3) When P finishes ($\checkmark$ happens), the control becomes Ω.

(Renaming 1) Whenever an event a happens in P, it is renamed to b ($a \,\Re\, b$) so that, externally, only b is visible. Renaming has no effect on events renamed to themselves ($a \,\Re\, a$), τ and $\checkmark$.

(Renaming 2) When P finishes ($\checkmark$ happens), the control becomes Ω.

We illustrate the semantics with the following example.

Example 2.1.1. Consider the next CSP specification:

$$\texttt{MAIN} \;=\; (\texttt{b} \rightarrow \texttt{STOP}) \;[\![\; \texttt{b} \;\Re\; \texttt{a} \;]\!] \underset{\{\texttt{a}\}}{\|} (\texttt{P} \;\Box\; (\texttt{b} \rightarrow \texttt{STOP}))$$

$$\texttt{P} \;=\; (\texttt{a} \rightarrow \texttt{SKIP}) \;;\; \texttt{STOP}$$

If we use `MAIN` as the initial state to execute the semantics, we get the computation shown in Fig. 2.3 where the final state is $\texttt{STOP} \;[\![\; \texttt{b} \;\Re\; \texttt{a} \;]\!] \underset{\{\texttt{a}\}}{\|} \texttt{STOP}$. This

(Process Call)

$$\frac{}{N \xrightarrow{\tau} rhs(N)}$$

(Parameterized Process Call)

$$\frac{}{N(\overline{a_n}) \xrightarrow{\tau} subs(\overline{a_n}, \overline{x_n}, rhs(N))}$$
where $N(\overline{x_n}) = rhs(N) \in \mathcal{S}$
with $\overline{x_n} \in \Sigma_{\mathcal{V}} \wedge \overline{a_n} \in \Sigma$

(Prefixing)

$$\frac{}{(co \rightarrow P) \xrightarrow{a} subs(a, co, P)} \quad a \in comms(co)$$

(SKIP)

$$\frac{}{\texttt{SKIP} \xrightarrow{\checkmark} \Omega}$$

(Internal Choice 1)

$$\frac{}{(P \sqcap Q) \xrightarrow{\tau} P}$$

(Internal Choice 2)

$$\frac{}{(P \sqcap Q) \xrightarrow{\tau} Q}$$

(External Choice 1)

$$\frac{P \xrightarrow{\tau} P'}{(P \,\Box\, Q) \xrightarrow{\tau} (P' \,\Box\, Q)}$$

(External Choice 2)

$$\frac{Q \xrightarrow{\tau} Q'}{(P \,\Box\, Q) \xrightarrow{\tau} (P \,\Box\, Q')}$$

(External Choice 3)

$$\frac{P \xrightarrow{e} P'}{(P \,\Box\, Q) \xrightarrow{e} P'} \quad e \in \Sigma \cup \{\checkmark\}$$

(External Choice 4)

$$\frac{Q \xrightarrow{e} Q'}{(P \,\Box\, Q) \xrightarrow{e} Q'} \quad e \in \Sigma \cup \{\checkmark\}$$

(Conditional Choice 1)

$$\frac{}{(P \trianglelefteq Bool \trianglerighteq Q) \xrightarrow{\tau} P} \text{ if } Bool = true$$

(Conditional Choice 2)

$$\frac{}{(P \trianglelefteq Bool \trianglerighteq Q) \xrightarrow{\tau} Q} \text{ if } Bool = false$$

(Sequential Composition 1)

$$\frac{P \xrightarrow{e} P'}{(P; Q) \xrightarrow{e} (P'; Q)} \quad e \in \Sigma \cup \{\tau\}$$

(Sequential Composition 2)

$$\frac{P \xrightarrow{\checkmark} \Omega}{(P; Q) \xrightarrow{\tau} Q}$$

(Synchronized Parallelism 1)

$$\frac{P \xrightarrow{e'} P'}{(P \underset{X}{||} Q) \xrightarrow{e} (P' \underset{X}{||} Q)} \quad \begin{array}{l}(e = e' \in \Sigma \backslash X) \;\vee \\ (e = \tau \;\wedge\; e' \in \{\tau, \checkmark\})\end{array}$$

(Synchronized Parallelism 2)

$$\frac{Q \xrightarrow{e'} Q'}{(P \underset{X}{||} Q) \xrightarrow{e} (P \underset{X}{||} Q')} \quad \begin{array}{l}(e = e' \in \Sigma \backslash X) \;\vee \\ (e = \tau \;\wedge\; e' \in \{\tau, \checkmark\})\end{array}$$

(Synchronized Parallelism 3)

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q'}{(P \underset{X}{||} Q) \xrightarrow{a} (P' \underset{X}{||} Q')} \quad a \in X$$

(Synchronized Parallelism 4)

$$\frac{}{(\Omega \underset{X}{||} \Omega) \xrightarrow{\checkmark} \Omega}$$

(Hiding 1)

$$\frac{P \xrightarrow{a} P'}{(P \backslash B) \xrightarrow{\tau} (P' \backslash B)} \quad a \in B$$

(Hiding 2)

$$\frac{P \xrightarrow{e} P'}{(P \backslash B) \xrightarrow{e} (P' \backslash B)} \quad (e \in \Sigma \wedge e \notin B) \;\vee\; (e = \tau)$$

(Hiding 3)

$$\frac{P \xrightarrow{\checkmark} \Omega}{(P \backslash B) \xrightarrow{\checkmark} \Omega}$$

(Renaming 1)

$$\frac{P \xrightarrow{e'} P'}{(P[\![\Re]\!]) \xrightarrow{e} (P'[\![\Re]\!])} \quad \begin{array}{l}(e, e' \in \Sigma \wedge e' \;\Re\; e) \;\vee \\ (e = e' = \tau)\end{array}$$

(Renaming 2)

$$\frac{P \xrightarrow{\checkmark} \Omega}{(P[\![\Re]\!]) \xrightarrow{\checkmark} \Omega}$$

Figure 2.2: CSP's operational semantics

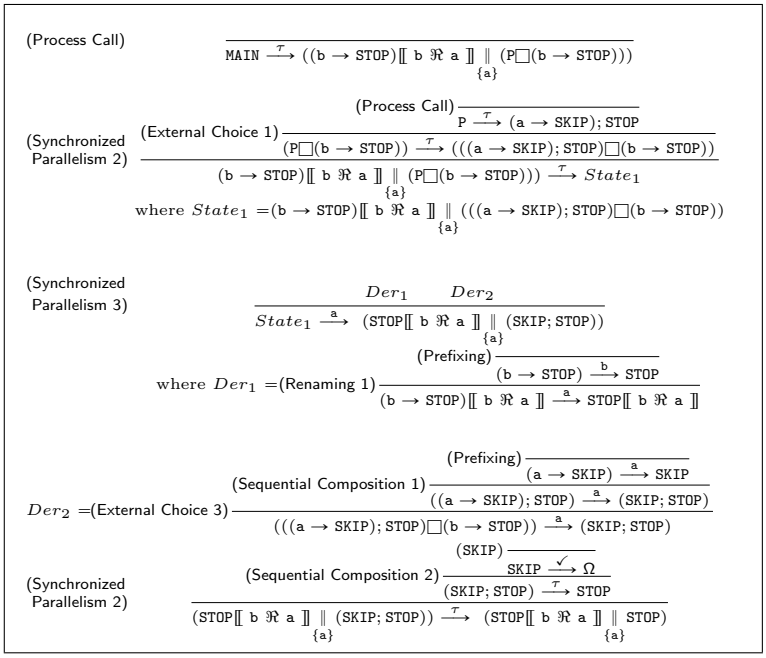


Figure 2.3: A computation with the operational semantics in Fig. 2.2

computation corresponds to the execution of the left branch of the choice (i.e., P) and thus event a occurs forcing a synchronization between both processes. Each rewriting step is labeled with the applied rule, and the example should be read from top to bottom.

2.2 Formal definition and basic terminology of Petri Nets

A Petri net [66, 72] is a directed bipartite graph, whose two essential elements are called *places* (represented by circles) and *transitions* (represented by bars or rectangles). The edges of the graph form the *arcs*, which are labelled with a positive integer known as *weight*. Arcs run from places to transitions and vice versa. The *state* of the system modeled by the net is represented by assigning non-negative integers to places. This is known as a *marking*, and is shown graphically by adding small black circles to the places, known as *tokens*. The *dynamic behavior* of the system is simulated by changes in the markings of a Petri net, a process which is carried out by the firing of the transitions. The basic concepts of Petri nets are summarized as follows:

2.2 Formal definition and basic terminology of Petri Nets

Definition 2.2.1. A *Petri net* [66, 72] is a tuple $\mathcal{N} = (P, T, F)$, where:

- P is a set of *places*.
- T is a set of *transitions*, such that $P \cap T = \varnothing$ and $P \cup T \neq \varnothing$.
- F is the *flow relation* that assigns weights to arcs: $F : P \times T \cup T \times P \to \mathbb{N}$.

The *marking* M of a Petri net is defined over the set of places P. For each place $p \in P$ we let $M(p)$ denote the number of tokens contained in p.

A *marked Petri net* Σ is a pair $(\mathcal{N}, M)$ where $\mathcal{N}$ is a Petri net and M is a marking. We denote by M_0 the *initial marking* of the net.

In the following, given a marking M and a set of places P, we denote by $M|_P$ the restriction of M over P, i.e., $M|_P(p) = M(p)$ for all $p \in P$ and $M|_P$ is undefined otherwise.

Definition 2.2.2. [72] Given a Petri net $\mathcal{N} = (P, T, F)$, we say that a marking M' *covers* a marking M if $M' \geqslant M$, i.e., $M'(p) \geqslant M(p)$ for each $p \in P$.

Given a Petri net $\mathcal{N} = (P, T, F)$, we say that a place $p \in P$ is an *input (resp. output) place* of a transition $t \in T$ iff there is an *input (resp. output) arc* from p to t (resp. from t to p). Given a transition $t \in T$, we denote by ${}^\bullet t$ and $t^\bullet$ the set of all input and output places of t, respectively. Analogously, given a place $p \in P$, we denote ${}^\bullet p$ and $p^\bullet$ the set of all input and output transitions of p, respectively.

Definition 2.2.3. Let $\Sigma = (\mathcal{N}, M)$ be a marked Petri net, with $\mathcal{N} = (P, T, F)$. We say that a transition $t \in T$ is *enabled* in M, in symbols $M \xrightarrow{t}$, iff for each input place $p \in P$ of t, we have $M(p) \geqslant F(p, t)$. A transition may only be fired if it is enabled.

The *firing* of an enabled transition t in a marking M eliminates $F(p, t)$ tokens from each input place $p \in {}^\bullet t$ and adds $F(t, p')$ tokens to each output place $p' \in t^\bullet$, producing a new marking M', in symbols $M \xrightarrow{t} M'$.

We say that a marking M_n is *reachable* from an initial marking M_0 if there exists a *firing sequence* $\sigma = t_1 t_2 \dots t_n$ such that $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \dots \xrightarrow{t_n} M_n$.

In this case, we say that M_n is reachable from M_0 through σ, in symbols $M_0 \xrightarrow{\sigma} M_n$. This notion includes the empty sequence ϵ; we have $M \xrightarrow{\epsilon} M$ for any marking M. We say that a firing sequence is *initial* if it starts from an initial marking.

The set of all possible markings which are reachable from an initial marking M_0 in a marked Petri net $\Sigma = (\mathcal{N}, M_0)$ is denoted by $R(\mathcal{N}, M_0)$ (or simply by $R(M_0)$ when $\mathcal{N}$ is clear from the context).

The following notion of *subnet* will be particularly relevant in the context of slicing (roughly speaking, we will identify a slice with a subnet). Let $P' \times T' \cup T' \times P' \subseteq P \times T \cup T \times P$, we say that a flow relation $F' : P' \times T' \cup T' \times P' \to \mathbb{N}$ is a restriction of another flow relation $F : P \times T \cup T \times P \to \mathbb{N}$ over P' and T', in symbols $F|_{(P',T')}$, if F' is defined as follows: $F'(x, y) = F(x, y)$ if $(x, y) \in P' \times T' \cup T' \times P'$ and F' is not defined otherwise.

Definition 2.2.4. [29] A *subnet* $\mathcal{N}' = (P', T', F')$ of a Petri net $\mathcal{N} = (P, T, F)$ is a Petri net such that $P' \subseteq P$, $T' \subseteq T$ and F' is a restriction of F over P' and T', i.e., $F' = F|_{(P',T')}$.

Chapter 3

Static Slicing of Explicitly Synchronized Languages

In this chapter we introduce a static analysis technique based on *program slicing* [88] for concurrent and explicitly synchronized languages in general, and CSP in particular. Our technique allows us to extract the part of a specification related to a given point (referred to as the slicing criterion) in the specification. This technique can be very useful to debug, understand, maintain and reuse specifications; but also as a preprocessing stage of other analyses and/or transformations in order to reduce the complexity of the specification. In particular, given a point (e.g., an event) in a specification, our technique allows us to extract those parts of the specification that must be executed before the specified point (thus they are an implicit precondition); and those parts of the specification that could be executed before it. Therefore, the other parts of the specification cannot be executed before this point.

Consider the specification of Figure 3.1. In this specification we have three processes (`STUDENT`, `PARENT` and `COLLEGE`) executed in parallel and synchronized on common events. Process `STUDENT` represents the three-year academic courses of a student; process `PARENT` represents the parent of the student who gives her a present when she passes a course; and process `COLLEGE` represents the college who gives a prize to those students that finish without any fail.

We are interested in determining what parts of the specification must be

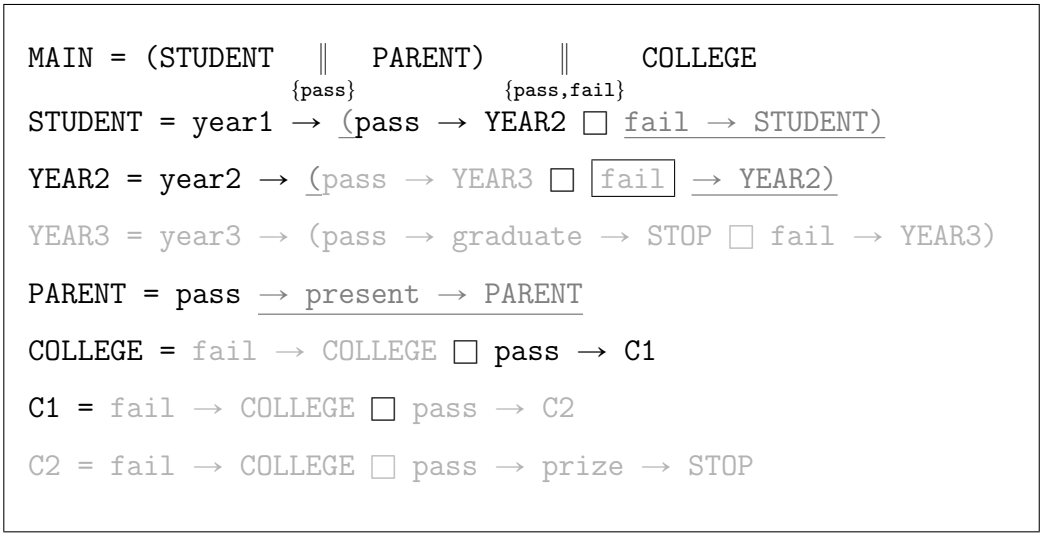


Figure 3.1: Example of a CSP specification

executed before the student fails in the second year, hence, we mark event `fail` of process `YEAR2` (thus the slicing criterion is `(YEAR2, fail)`, marked by a box in Figure 3.1). Our slicing technique automatically extracts the slice consisting of the expressions in black. We can additionally be interested in knowing what parts could be executed before the same event. In this case, our technique adds to the slice the underscored parts because they could be executed (in some executions) before the marked event (observe that the result of this analysis is always a superset of the result obtained by the previous analysis). Therefore, this analysis could be used for program comprehension. Note, for instance, that in order to fail in the second year, the student has necessarily passed the first year. But, the parent may or may not have given a present to his daughter (even if she passed the first year) because this specification does not force the parent to give a present to his daughter until she has passed the second year. Moreover, note that the choice of process `C1` belongs also to the slice. This is due to the fact that the slicing criterion must synchronize with the event `fail` of this process; therefore, the choice must be executed before the slicing criterion.[1] This is not so obvious from the specification, and the slice can help to understand the actual meaning of the specification.

Computing the parts of the specification that could be executed before the

[1] We could have chosen also to include the `fail` event of `C1` into the slice. This is definitely a design decision.

slicing criterion can be useful, e.g., for debugging. If the slicing criterion is an event that was executed incorrectly (i.e., it should not happen), then the slice produced contains all the parts of the specification that could produce the wrong behavior.

A third application is program specialization. Note that the slices produced are not executable, but, in both cases, the slices could be made executable by replacing the removed parts by "STOP" or by "→ STOP" if the removed expression has a prefix. Hence, we have defined a further transformation that allows us to extract executable slices. The specialized specification contains all the necessary parts of the original specification whose execution leads to the slicing criterion (and then, the specialized specification finishes).

It should be clear that computing the minimum slice of an arbitrary CSP specification is an undecidable problem. Consider for instance the following CSP specification:

```
MAIN = P ⊓ Q

P = X ; Q

Q = a → STOP

X = Infinite Process
```

together with the slicing criterion (Q, a). Determining whether X does not belong to the slice implies determining whether X terminates, which is undecidable.

Our technique is based on a new data structure that extends the *Synchronized Control Flow Graph* (SCFG). We show that this new data structure improves the SCFG by taking into account the context in which processes are called and, thus, it makes the slicing process more precise. In Paper 1 we define a data structure called *Context-sensitive Synchronized Control-Flow Graph* (CSCFG) that allows us to statically simplify a specification before the analyses. This simplification is automatic and thus it is very useful as a preprocessing stage of other analyses. The CSCFG is a graph that allows us to finitely represent possibly infinite computations, and it is particularly interesting because it takes into account the context of process calls, and thus it allows us to produce analyses that are very precise.

However, computing the CSCFG is a complex task due to the non-deterministic execution of processes, deadlocks, non-terminating processes and synchronizations. We present a correctness result which formally relates the CSCFG of a specification to its execution. This result is needed to prove important properties (such as correctness and completeness) of the techniques based on the CSCFG.

We also introduce an algorithm able to automatically generate this data structure from a CSP specification. We formally define the CSCFG and a technique to produce the CSCFG of a given CSP specification. In Paper 2 we introduce a new formalization of the CSCFG that directly relates the graph construction to the control-flow of the computations it represents. Roughly, we instrument the CSP standard semantics (Chapter 7 in [81]) in such a way that the execution of the instrumented semantics produces as a side-effect the portion of the CSCFG associated with the performed computation. Then, we define an algorithm which uses the instrumented semantics to build the complete CSCFG associated with a CSP specification. This algorithm executes the semantics several times to explore all possible computations of the specification, producing incrementally the final CSCFG. Algorithms to construct CSCFGs have been implemented and integrated into the most advanced CSP environment ProB [49].

The algorithm has been proved correct. We state the correctness of the proposed algorithm by showing that (i) the graph produced by the algorithm for a CSP specification is its CSCFG; and (ii) the algorithm terminates, even if non-terminating computations exist for a specification.

The technique has been implemented and tested with real specifications, producing good results. We describe our tool, its architecture, its main applications and the results obtained from several experiments conducted in order to measure the performance of the tool. The result is the first program slicer for CSP specifications. In our implementation, the slicing process is completely automatic. Once the user has loaded a specification, she can select (with the mouse) the point she is interested in. Obviously, this simple action is enough to define a slicing criterion because the tool can automatically determine the process and the source position of interest. This implementation is a tool that has been integrated in the system ProB [49, 16], an animator and model checker for B and CSP.

3.1 Context-sensitive Synchronized Control Flow Graph

As usual in static analysis, we need a data structure capable of finitely representing the (often infinite) computations of our specifications. Unfortunately, we cannot use the standard *Control Flow Graph* (CFG) [86], nor the *Interprocedural Control Flow Graph* (ICFG) [35] because they cannot represent multiple threads and, thus, they can only be used with sequential programs. In fact, for CSP specifications, being able to represent multiple threads is a necessary but not a sufficient condition. For instance, the *threaded Control Flow Graph* (tCFG) [45, 46] can represent multiple threads through the use of the so called "*start thread*" and "*end thread*" nodes; but it does not handle synchronization between threads. Callahan and Sublok introduced in [17] the *Synchronized Control Flow Graph* (SCFG), a data structure proposed in the context of imperative programs where an event variable is always in one of two states: clear or posted. The initial value of an event variable is always clear. The value of an event variable can be set to posted with the *POST* statement; and a *WAIT* statement suspends execution of the thread that executes it until the specified event variable value is set to posted. The SCFG explicitly represents synchronization between threads with a special edge for synchronization flows. According to Callahan and Sublok [17]:

> "*A synchronized control flow graph is a control flow graph augmented with a set E_s of synchronization edges. $(b_1, b_2) \in E_s$ if the last statement in block b_1 is $POST(ev)$ and the first statement in block b_2 is $WAIT(ev)$ where ev is an event variable.*"

This chapter and the next one use labels for the CSP components wich we call *specification positions*.[2] Consider the CSP specification in Fig. 3.2 where literals are labelled with their associated specification positions (they are underlined) so that labels are unique

In order to adapt the SCFG to CSP, we extend it with the "*start thread*" and "*end thread*" notation from tCFGs. Therefore, in the following we will work with graphs where nodes N are labeled with positions and "*start*", "*end*" labels. We also use this notation, "*end* $\backslash$" and "*end* $[\![]\!]$", to denote the end of a hiding, respectively a renaming operator.

[2] In Section 2 of Paper 1 the reader can find a complete description of how this labeling process is performed.

$$\texttt{MAIN}_{(\texttt{MAIN},0)} = (\texttt{BUS}_{(\texttt{MAIN},1.1)} ||_{(\texttt{MAIN},1)} \texttt{P1}_{(\texttt{MAIN},1.2)}) ;_{(\texttt{MAIN},\Lambda)} (\texttt{BUS}_{(\texttt{MAIN},2.1)} ||_{(\texttt{MAIN},2)} \texttt{P2}_{(\texttt{MAIN},2.2)})$$

$$\texttt{BUS}_{(\texttt{BUS},0)} = \texttt{board}_{(\texttt{BUS},1)} \rightarrow_{(\texttt{BUS},\Lambda)} \texttt{alight}_{(\texttt{BUS},2.1)} \rightarrow_{(\texttt{BUS},2)} \texttt{SKIP}_{(\texttt{BUS},2.2)}$$

$$\texttt{P1}_{(\texttt{P1},0)} = \texttt{wait}_{(\texttt{P1},1)} \rightarrow_{(\texttt{P1},\Lambda)} \texttt{board}_{(\texttt{P1},2.1)} \rightarrow_{(\texttt{P1},2)} \texttt{alight}_{(\texttt{P1},2.2.1)} \rightarrow_{(\texttt{P1},2.2)} \texttt{SKIP}_{(\texttt{P1},2.2.2)}$$

$$\texttt{P2}_{(\texttt{P2},0)} = \texttt{wait}_{(\texttt{P2},1)} \rightarrow_{(\texttt{P2},\Lambda)} \texttt{board}_{(\texttt{P2},2.1)} \rightarrow_{(\texttt{P2},2)} \texttt{pay}_{(\texttt{P2},2.2.1)} \rightarrow_{(\texttt{P2},2.2)} \texttt{alight}_{(\texttt{P2},2.2.2.1)} \rightarrow_{(\texttt{P2},2.2.2)} \texttt{SKIP}_{(\texttt{P2},2.2.2.2)}$$

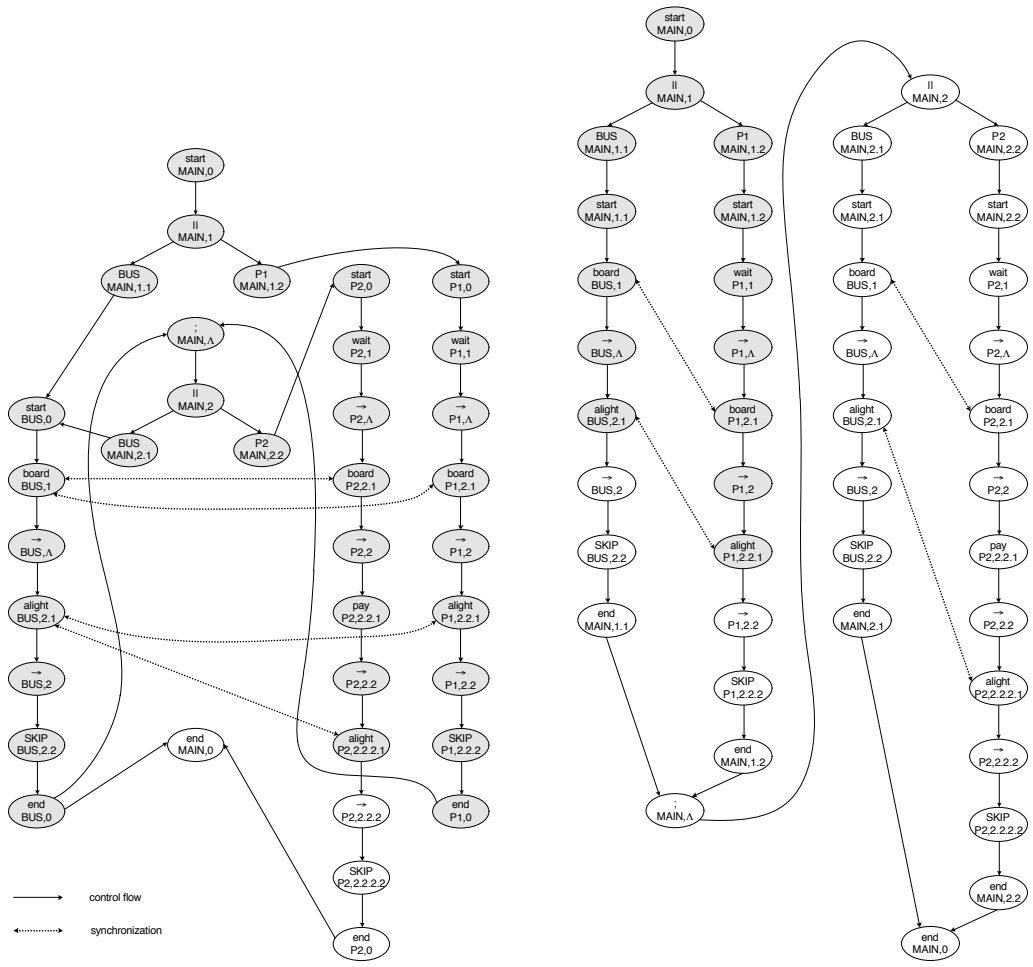
Figure 3.2: Simplification of a benchmark to simulate a bus line

Given a CSP specification $\mathcal{S}$, we define its *Synchronized Control Flow Graph* as a graph $\mathcal{G} = (N, E_c, E_s)$ where nodes in N are specification positions or *start* and *end* nodes. Edges are divided into two groups, *control-flow arcs* (E_c) and *synchronization edges* (E_s). E_s is a set of edges (drawn as dotted arrows) representing the possible synchronization of two (event) nodes. E_c is a set of arcs (drawn as plain arrows) representing the control flow.

There is only one node in the SCFG for each position of the specification, and specification positions are finite and unique. Therefore, the size of the SCFG is $\mathcal{O}(n)$ being n the number of positions in the specification. To be fully precise, there is exactly one node for each specification position and two extra nodes for each process (the start process and end process nodes) and one extra node for the hiding and renaming operators (the end hiding and the end renaming). Hence, the size of a SCFG associated to a specification with p processes and n positions with r hiding and renaming operators is $2p + n + r$.

The SCFG can be used for slicing CSP specifications. Consider the specification of Figure 3.2. It is a simplification of a benchmark by Simon Gay to simulate a bus line. Its associated SCFG is shown in Figure 3.3(a); for the sake of clarity we show the expression represented by each specification position. If we select the node labeled `(P1,alight)` and traverse the SCFG backwards in order to identify the nodes on which `(P1,alight)` depends, we get the grey nodes of the graph.

The purpose of this example is twofold: on the one hand, it shows that the SCFG can be used for static slicing of CSP specifications. On the other hand, it shows that it is still too imprecise to be used in practice. The cause of this imprecision is that the SCFG is context-insensitive, because it connects all the calls to the same process with a unique set of nodes. This causes



(a) SCFG (b) CSCFG

Figure 3.3: SCFG and CSCFG of the program in Figure 3.2

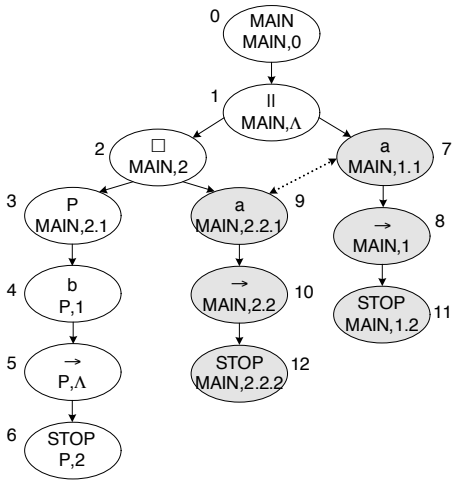
the SCFG to mix different executions of a process with possibly different synchronizations, and, thus it loses precision. For instance, in the CSP specification of Figure 3.2 process `BUS` is called twice in different contexts. It is first executed in parallel with `P1` producing the synchronization of their `board` and `alight` events. Then, it is executed in parallel with `P2` producing the synchronization of their `board` and `alight` events. This makes the process `P2` (except nodes →, `SKIP` and `end P2`) be part of the slice. This is suboptimal because process `P2` is always executed after `P1`.

To the best of our knowledge, there do not exist other data structures that face the problem of representing concurrent and explicitly synchronized com-

$$\texttt{MAIN}_{(\texttt{MAIN},0)} = (\texttt{a}_{(\texttt{MAIN},1.1)} \rightarrow {}_{(\texttt{MAIN},1)}\texttt{STOP}_{(\texttt{MAIN},1.2)}) \underset{\{\texttt{a}\}}{\parallel}_{(\texttt{MAIN},\Lambda)}$$
$$(\texttt{P}_{(\texttt{MAIN},2.1)} \Box_{(\texttt{MAIN},2)} (\texttt{a}_{(\texttt{MAIN},2.2.1)} \rightarrow {}_{(\texttt{MAIN},2.2)}\texttt{STOP}_{(\texttt{MAIN},2.2.2)}))$$

$$\texttt{P}_{(\texttt{P},0)} = \texttt{b}_{(\texttt{P},1)} \rightarrow {}_{(\texttt{P},\Lambda)}\texttt{SKIP}_{(\texttt{P},2)}$$

(a) CSP specification



(b) CSCFG

Figure 3.4: CSP specification and its associated CSCFG

putations in a context-sensitive manner. In the rest of this section, we propose a new version of the SCFG, the context-sensitive synchronized control flow graph (CSCFG) which is context-sensitive because it takes into account the different contexts on which a process can be executed. Intuitively speaking, the context of a node represents the set of processes in which a particular node is being executed. This is represented by the set of process calls in the computation that were done before the specified node.

For instance, the CSCFG associated with the specification in Figure 3.4(a) is shown in Fig. 3.4(b). In this graph we have that $\mathcal{C}on(4) = \{0, 3\}$, i.e., `b` is being executed after having called processes `MAIN` and `P`. Focussing on a process call node we can use the context to identify loops.

In contrast to the SCFG, the same specification position can appear multiple times inside a CSCFG. Hence, in the following we will use a refined notion of the "*start thread*" and "*end thread*" so that in each "*start* α" and "*end* α" node used to represent a process, α is now any specification position representing a process call instead of a process definition. Using the specification position of the process call allows us to distinguish between different process calls to the same process.

The main difference between the SCFG and the CSCFG is that the SCFG represents a process with a single collection of nodes (each specification position in the process is represented with a single node, see Figure 3.3(a)); in contrast, the CSCFG represents a process with multiple collections of nodes, each collection representing a different call to this process (i.e., a different context in which it is executed. For instance, see Figure 3.3(b) where process `BUS` is represented twice). Therefore, the notion of control flow used in the SCFG is insufficient for the CSCFG, and we need to extend it to also consider the context of process calls. Additionally, we add a new set of arcs (E_l) that represents loops.

Therefore, each process call is connected (with a control-flow edge) to a subtree which contains the right-hand side of the called process. Each subtree is a new subgraph except if a loop is detected. Consider again the specification of Fig. 3.4(a) and its associated CSCFG, shown in Fig. 3.4(b). For the time being, the reader can ignore the color of the nodes; they will be explained in Section 3.1.1. Each process call is connected to a subgraph which contains the right-hand side of the called process. For convenience, in this example there are no loop edges; there are control-flow edges and one synchronization edge between nodes (`MAIN`, `2.2.1`) and (`MAIN`, `1.1`) representing the synchronization of event `a`.

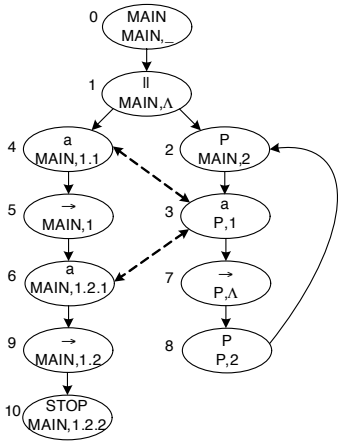
Loop edges allow us to finitely represent infinite computations. They are used when the same process call appears twice in a path starting from `MAIN`, and the first process has not been terminated. Consider the CSP specification of Figure 3.5(a). This specification can produce the sequence of events {`a`, `a`} and its associated CSCFG is shown in Fig. 3.5(b), where there are a loop edge from node 8 to node 2 and two synchronization edges between nodes 4 and 3 and nodes 6 and 3.

Another important difference between the SCFG and the CSCFG is that the latter unfolds every process call node except those that belong to a loop. This is very convenient for slicing because every process call that is executed in a different context is unfolded and represented with a different subgraph,

$$\texttt{MAIN} = \texttt{a}_{(\texttt{MAIN},1.1)} \rightarrow {}_{(\texttt{MAIN},1)}\texttt{a}_{(\texttt{MAIN},1.2.1)} \rightarrow {}_{(\texttt{MAIN},1.2)}\texttt{STOP}_{(\texttt{MAIN},1.2.2)} \underset{\{\texttt{a}\}}{\parallel}{}_{(\texttt{MAIN},\Lambda)}\texttt{P}_{(\texttt{MAIN},2)}$$

$$\texttt{P} = \texttt{a}_{(\texttt{P},1)} \rightarrow {}_{(\texttt{P},\Lambda)}\texttt{P}_{(\texttt{P},2)}$$

(a) CSP specification



(b) CSCFG

Figure 3.5: CSP specification with a looped process and its associated CSCFG

thus, slicing does not mix computations. Moreover, it allows us to deal with recursion and, at the same time, it prevents infinite unfolding of process calls thanks to the loop arcs that are used when the context is repeated. Loops are unfolded only once because the second time they are going to be unfolded the context of the process call node is repeated, and thus a loop arc is used to prevent the unfolding. Process calls only have one outgoing arc, and thus, they cannot have a control arc if there is already a loop arc. This property ensures finiteness.

We state that the CSCFG is complete because all possible derivations of a CSP specification $\mathcal{S}$ are represented in the CSCFG associated to $\mathcal{S}$. Of course, because it is a static representation of any possible execution, the CSCFG also introduces a source of imprecision. This imprecision happens when loop arcs are introduced in a CSCFG, because a loop arc summarizes

the rest of a computation with a single collection of nodes, and this collection could mix synchronizations of different iterations. However, note that all process calls of the specification are unfolded and represented with an exclusive collection of nodes, and loop arcs are only introduced if the same call is repeated again. This produces a high level of precision for slicing algorithms.

Note that the CSCFG shows the exact processes that have been evaluated with an explicit causality relation; and, in addition, it shows the specification positions that have been evaluated and in what order. Therefore, it is not only useful as a program comprehension tool, but it can be used for program simplification. For instance, with a simple backwards traversal from `a`, the CSCFG reveals that the only part of the code that can be executed before `a` is the underlined part:

$$\underline{\texttt{MAIN}} = (\underline{\texttt{a}} \rightarrow \texttt{STOP}) \underset{\underline{\{\texttt{a}\}}}{\parallel} (\texttt{P} \,\underline{\Box}\, (\underline{\texttt{a}} \rightarrow \texttt{STOP}))$$

$$\texttt{P} = \texttt{b} \rightarrow \texttt{STOP}$$

Hence, the specification can be significantly simplified for those analyses focussing on the occurrence of event `a`.

Apart from a declarative definition, we also provide a constructive method for the CSCFG that is the basis of our implementation. In particular, the CSCFG can be constructed starting from `MAIN`, and connecting each process call to a subgraph that contains the right-hand side of the process called . Each right-hand side is a new subgraph except if a loop is detected.

3.1.1 An Algorithm to generate the CSCFG

This section introduces an algorithm which is able to generate the CSCFG associated to a CSP specification. The algorithm uses an instrumented operational semantics of CSP which (i) generates as a side-effect the CSCFG associated to the computation performed with the semantics; (ii) controls that no infinite loops are executed; and (iii) ensures that the execution is deterministic.

Algorithm 3.1 controls that the semantics is executed repeatedly in order to deterministically execute all possible computations—of the original (non-deterministic) specification—and the CSCFG for the whole specification is

Algorithm 3.1 General Algorithm

Build the initial state of the semantics:
$state \;=\; (\texttt{MAIN}_{(\texttt{MAIN},0)}, \varnothing, \bullet, (\varnothing, \varnothing), \varnothing, \varnothing)$
repeat
  repeat
    Run the rules of the instrumented semantics with the state *state*
  until no more rules can be applied
  Get the new state: $state \;=\; (\_, G, \_, (\varnothing, S_0), \_, \zeta)$
  $state = (\texttt{MAIN}_{(\texttt{MAIN},0)}, G, \bullet, (\texttt{UpdStack}(S_0), \varnothing), \varnothing, \varnothing)$
until $\texttt{UpdStack}(S_0) = \varnothing$
return G
where function `UpdStack` is defined as follows:

$$\texttt{UpdStack}(S) = \begin{cases} (rule, rules \backslash \{rule\}) : S' & \text{if } S = (\_, rules) : S' \text{ and } rule \in rules \\ \texttt{UpdStack}(S') & \text{if } S = (\_, \varnothing) : S' \\ \varnothing & \text{if } S = \varnothing \end{cases}$$

constructed incrementally with each execution of the semantics. The key point of the algorithm is the use of a stack that records the actions that can be performed by the semantics. In particular, the stack contains tuples of the form $(rule, rules)$ where $rule$ indicates the rule that must be selected by the semantics in the next execution step, and $rules$ is a set with the other possible rules that can be selected. The algorithm uses the stack to prepare each execution of the semantics indicating the rules that must be applied at each step. For this, is is used function `UpdStack` that basically avoids to repeat the same computation with the semantics. When the semantics finishes, the algorithm prepares a new execution of the semantics with an updated stack. This is repeated until all possible computations are explored (i.e., until the stack is empty). Additionally, we have to consider that the standard operational semantics of CSP [81] can be non-terminating due to infinite computations. Therefore, the instrumentation of the semantics incorporates a loop-checking mechanism to ensure termination.

The instrumented semantics is an operational semantics where we assume that every literal in the specification has been labelled with its specification position. In this semantics, a *state* is a tuple $(P, G, m, (S, S_0), \Delta, \zeta)$, where P is the process to be evaluated (the *control*), G is a directed graph (i.e., the CSCFG constructed so far), m is a numeric reference to the current node in G, (S, S_0) is a tuple with two stacks that contains the rules to apply and the rules applied so far, Δ is a set of references to nodes used to draw

synchronizations in G, and ζ is a graph like G, but it only contains the part of the graph generated for the current computation, and it is used to detect loops. The basic idea of the graph construction is to record the current control with a fresh reference n by connecting it to its parent m.

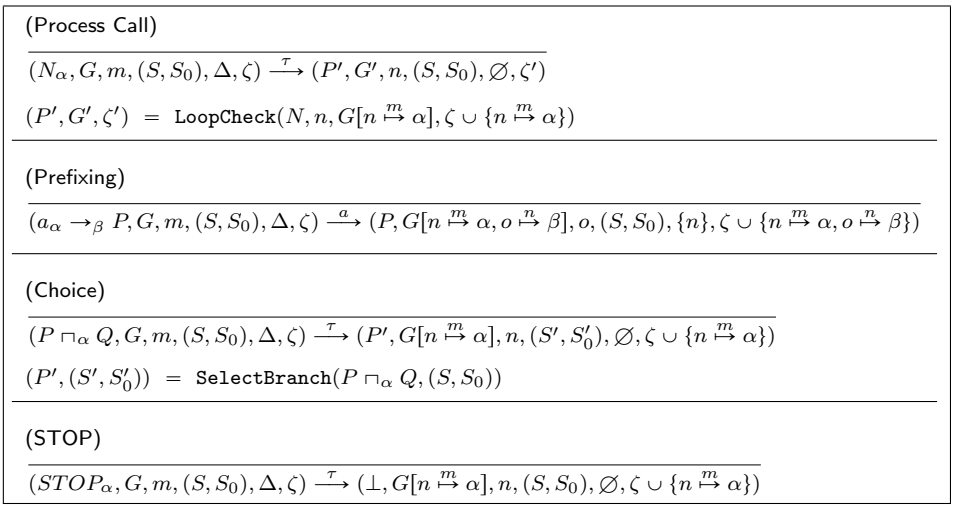


Figure 3.6: An instrumented operational semantics that generates the CSCFG

The complete instrumented semantics is presented in Figure 3.6. A brief explanation for the most significant rules of the semantics follows.

(Process Call) The called process is unfolded and a node with a fresh reference is added to the graph. The new expression in the control is computed with function `LoopCheck` that prevents infinite unfolding. Intuitively, this function checks whether the process call in the control has not been already executed (if so, we are in a loop). When a loop is detected, a loop edge is added to the graph; and the right-hand side of the called process is labelled with a special symbol $\circlearrowleft_s$. This label is later used by rule (Synchronized Parallelism 4) to decide whether the process must be stopped. The loop symbol $\circlearrowleft$ is labelled with the position s of the process call of the loop. This is used to know what is the reference of the process' node if it is unfolded again.

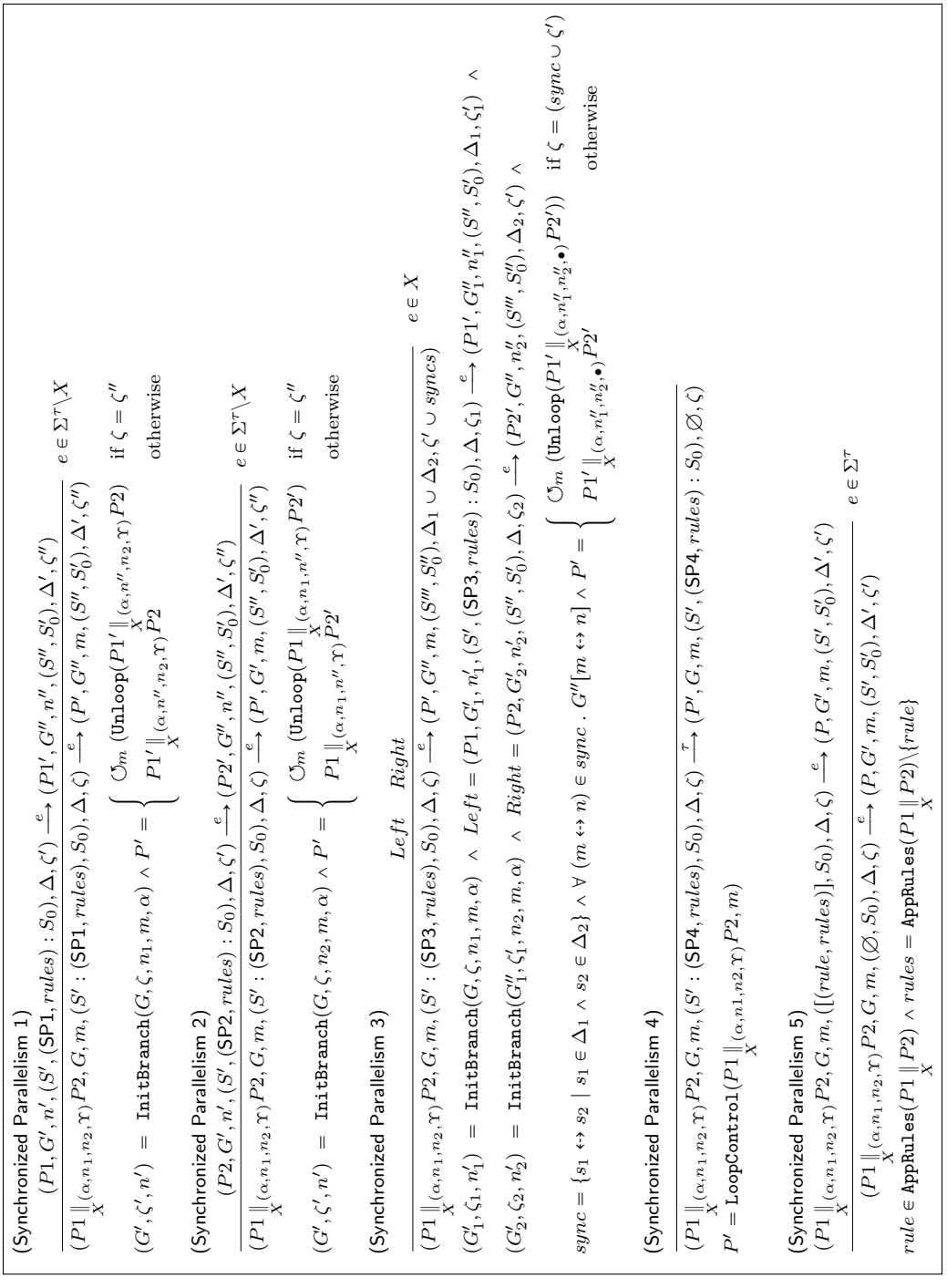


Figure 3.6: An instrumented operational semantics that generates the CSCFG (cont.)

(Choice) The only sources of non-determinism are choice operators (different branches can be selected for execution) and parallel operators (different order of branches can be selected for execution). Therefore, every time the semantics executes a choice or a parallelism, they are made deterministic thanks to the information in the stack S. Both internal and external can be treated with a single rule because the CSCFG associated to a specification with external choices is identical to the CSCFG associated to the specification with the external choices replaced by internal choices.

Function `SelectBranch` is used to produce the new control and the new tuple of stacks by selecting a branch using the information of current stacks. Given a choice $P \sqcap Q$ and stacks (S, S_0), if the last element of the stack S indicates that the first branch of the choice (C1) must be selected, then P is the new control. If the second branch must be selected (C2), the new control is Q. In any other case the stack is empty, and thus this is the first time that this choice is evaluated. Then, we select the first branch (P is the new control) and we add $(\mathsf{C1}, \{\mathsf{C2}\})$ to the stack S_0 indicating that C1 has been fired, and the remaining option is C2.

For instance, when the CSCFG of Fig. 3.4(b) is being constructed and we reach the choice operator (i.e., $(\texttt{MAIN}, 2)$), then the left branch of the choice is evaluated and $(\mathsf{C1}, \{\mathsf{C2}\})$ is added to the stack to indicate that the left branch has been evaluated. The second time it is evaluated, the stack is updated to $(\mathsf{C2}, \varnothing)$ and the right branch is evaluated. Therefore, the selection of branches is predetermined by the stack, thus, the algorithm can decide what branches are evaluated by conveniently handling the information of the stack.

(Synchronized Parallelism 1 and 2) The stack determines what rule to use when a parallelism operator is in the control. If the last element in the stack is SP1, then (Synchronized Parallelism 1) is used. If it is SP2, (Synchronized Parallelism 2) is used.

In a synchronized parallelism composition, both parallel processes can be intertwiningly executed until a synchronized event is found. Therefore, nodes for both processes can be added interwoven to the graph. Hence, the semantics needs to know in every state the references to be used in both branches. This is done by labelling each parallelism operator with a tuple of the form $(\alpha, n_1, n_2, \Upsilon)$ where α is the specification position of the parallelism operator; n_1 and n_2 are respectively the references of the last node introduced in the left and right branches of the parallelism, and they are initialised to $\bullet$; and Υ is a node reference used to decide when to unfold a process call (in order to avoid infinite loops), also initialised to $\bullet$. If the graph of the

current computation remains the same, meaning that nothing has change in this derivation, this rule detects that the parallelism is in a loop; and thus, in the new control the parallelism operator is labelled with $\circlearrowleft$ and all the other loop labels are removed from it.

These rules develop the branches of the parallelism until they are finished or until they must synchronize. They introduce the parallelism into the graph the first time it is executed and only if it has not been introduced in a previous computation. For instance, consider a state where a parallelism operator is labelled with $((\texttt{MAIN}, \Lambda), \bullet, \bullet, \bullet)$. Therefore, it is evaluated for the first time, and thus, when, e.g., rule (Synchronized Parallelism 1) is applied, a node which refers to the parallelism operator, is added to the graph and the parallelism operator is relabelled to $((\texttt{MAIN}, \Lambda), \texttt{x}, \bullet, \bullet)$ where x is the new reference associated with the left branch.

(Synchronized Parallelism 3) It is applied when the last element in the stack is SP3. It is used to synchronize the parallel processes. In this rule, Υ is replaced by $\bullet$, meaning that a synchronization edge has been drawn and the loops could be unfolded again if it is needed. All the events that have been executed in this step must be synchronized. Therefore, all the events occurred in the subderivations of the paralleled processes are mutually synchronized and added to the graph.

(Synchronized Parallelism 4) This rule is applied when the last element in the stack is SP4. It is used when none of the parallel processes can proceed (because they already finished, deadlocked or were labelled with $\circlearrowleft$). When a process is labelled as a loop with $\circlearrowleft$, it can be unlabelled to unfold it once[3] in order to allow the other processes to continue. This happens when the looped process is in parallel with other process and the later is waiting to synchronize with the former. In order to perform the synchronization, both processes must continue, thus the loop is unlabelled. Hence, the system must stop only when both parallel processes are marked as a loop. This task is done by function `LoopControl` that decides if the branches of the parallelism should be further unfolded or they should be stopped (e.g., due to a deadlock or an infinite loop).

When one of the branches has been labelled as a loop, there are three options: (i) The other branch is also a loop. In this case, the whole parallelism is marked as a loop labelled with its parent, and Υ is put to $\bullet$. (ii) Either it

[3]Only once because it will be labelled again by rule (Process Call) when the loop is repeated.

is a loop that has been unfolded without drawing any synchronization (this is known because Υ is equal to the parent of the loop), or the other branch already terminated (i.e., it is $\perp$). In this case, the parallelism is also marked as a loop, and the other branch is put to $\perp$ (this means that this process has been deadlocked). Also here, Υ is put to $\bullet$. (iii) If we are not in a loop, then we allow the parallelism to proceed by unlabelling the looped branch. When none of the branches has been labelled as a loop, $\perp$ is returned representing that this is a deadlock, and thus, stopping further computations.

(Synchronized Parallelism 5) This rule is used when the stack is empty. It basically analyses the control and decides what are the set of rules that can be applied to a synchronized parallelism.

Essentially, it decides what rules are applicable depending on the events that could happen in the next step. These events can be inferred by using function `AppRules` that, given a process P, returns the set of events that can fire a rule in the semantics using P as the control. Therefore, rule (Synchronized Parallelism 5) prepares the stack allowing the semantics to proceed with the correct rule.

Consider again the specification in Figure 3.4(a). Due to the choice operator, in this specification two different events can occur, namely `b` and `a`. Therefore, the algorithm performs two iterations (one for each computation) to generate the final CSCFG. Figure 3.4(b) shows the CSCFG generated where white nodes were produced in the first iteration; and grey nodes were produced in the second iteration.

3.1.2 Using the CSCFG for Program Slicing

For slicing purposes, the CSCFG is interesting because we can use the edges to determine if a node must be executed or not before another node, thanks to the following properties:

- if a control edge exists from n to n' then n must be executed before n' in all executions.
- if a loop edge exists from n to n' then n' must be executed before n in all executions.
- if a synchronization edge exists between n and n' then n and n' are executed at the same time in all executions.

Thanks to the fact that loops are unfolded only once, the CSCFG ensures that all the specification positions inside the loops are in the graph and can be collected by slicing algorithms. For slicing purposes, this representation also ensures that every possibly executed part of the specification belongs to the CSCFG because only loops (i.e., repeated nodes) are missing.

Consider the specification of Figure 3.2 and its associated CSCFG shown in Figure 3.3(b). If we select the node labeled `(P1,alight)` and traverse the CSCFG backwards in order to identify the nodes on which this node depends, we only get the nodes of the graph colored in gray. This particular slice is optimal and much smaller than the slice obtained when we select the same node `(P1,alight)` in the SCFG (see Figure 3.3(a)).

The CSCFG provides a different representation for each context in which a process call is made. This can be seen in Figure 3.3(b) where process `BUS` appears twice to account for the two contexts in which it is called. In particular, in the CSCFG we have a fresh node to represent each different process call, and two nodes point to the same process if and only if they are the same call (they are labeled with the same specification position) and they belong to the same loop. This property ensures that the CSCFG is finite.

The specification in Figure 3.7 makes clear the difference between the SCFG and the CSCFG. While the SCFG only uses one representation for the process `P` (there is only one `start P`), the CSCFG uses four different representations because `P` could be executed in four different contexts. Note that due to the infinite loops, some parts of the graph are not reachable from `start MAIN`; i.e., there is no possible control flow to `end MAIN`.

3.2 Static Slicing of CSP specifications

We want to perform two kinds of analysis. Given a point in the specification, we want, on the one hand, to determine what parts of the specification MUST be executed before (MEB) it (in every possible execution); and, on the other hand, we want to determine what parts of the specification COULD be executed before (CEB) it (in any possible execution). Both analyses are closely related but they must be computed differently. While MEB is mainly based on backward slicing, CEB is mainly based on forward slicing to explore what could be executed in parallel processes.

In our approach the slicing criterion is a specification position. Clearly,

```
MAIN = P ; P

P = Q

Q = P
```

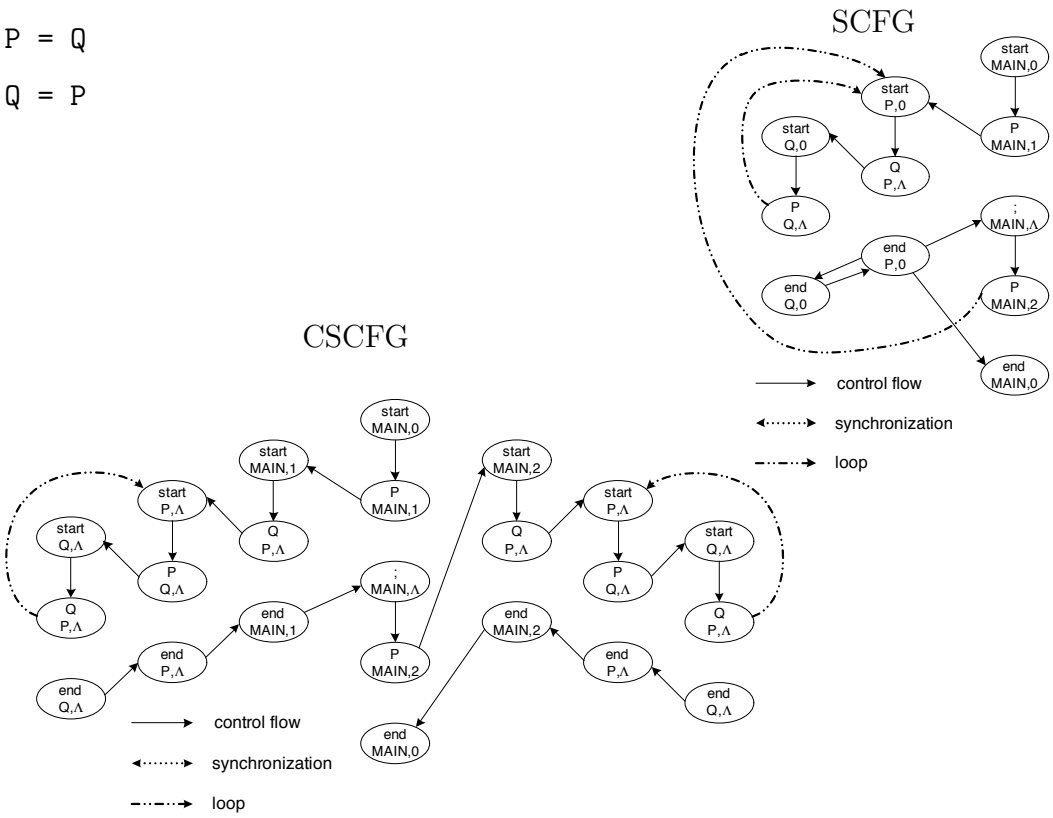


Figure 3.7: SCFG and CSCFG representing an infinite computation

the slicing criterion points to a set of nodes in the CSCFG, because the same specification position can happen in different contexts and, thus, it is represented in the CSCFG with different nodes. This means that a slicing criterion $\mathcal{C}$ is used to produce a slice with respect to *all* possible executions of $\mathcal{C}$. As an example, consider the slicing criterion `(BUS,alight)` for the specification in Figure 3.2, and observe in its CSCFG in Figure 3.3(b) that two different nodes are identified by the slicing criterion.

Note that the slicing criterion could point to nodes that are not reachable from `MAIN` such as dead code (see, e.g., Figure 3.7). Therefore, we exclude these nodes so that only feasible computations (starting from `MAIN`) are considered. Moreover, the slicing criterion could also point to different nodes that represent the same specification position that is executed many times in a (sub)computation (see, e.g., specification position $(\texttt{P}, \Lambda)$ in the CSCFG of Figure 3.7). Thus, we only select the first occurrence of this specification

position in the computation.

Given a slicing criterion, we use the CSCFG to approximate MEB and CEB. Computing correct slices is known as an undecidable problem even in the sequential setting (see, e.g., [88]). Therefore, our MEB and CEB analyses are an over-approximation. Technical results that ensure the completeness of the analyses can be found in Paper 1.

Regarding the MEB analysis, one could consider that a simple backwards traversal of the graph from the slicing criterion's nodes would produce a correct slice. However, this would produce a rather imprecise slice because this would include both branches of the choices in the path from `MAIN` to to the slicing criterion even if they do not need to be executed before it (consider for instance the process `((a→SKIP)□(b→SKIP));P` and the slicing criterion `P`). The union of paths from `MAIN` to the slicing criterion's nodes is not a solution, either, because it would be too imprecise by including in the slice parts of code that are executed before the slicing criterion only in some executions. For instance, in the process `(b→a→SKIP)□(c→a→SKIP)`, `c` belongs to one of the paths to `a`, but it must be executed before `a` or not depending on the choice. The intersection of paths is not a solution, either, as it can be seen in the process `a→((b→SKIP)||(c→SKIP));P` where `b` must be executed before `P`, but it does not belong to all the paths from `MAIN` to `P`.

In Paper 1 we formally define the notion of MEB slice and introduce an algorithm that can be used to compute the MEB analysis. It basically computes for each node in slicing criterion's nodes a set containing the part of the specification that must be executed before it. Then, it returns MEB as the intersection of all these sets. Each set, which is represented by *Meb*, is computed with an iterative process that takes a node and performs the following actions:

1. It starts with an initial set of nodes computed by collecting those nodes that were executed just before the initial node (i.e., they are connected to it or to a node synchronized with it with a control arc).

2. The initial set *Meb* is the backwards traversal of the CSCFG from the initial set following control arcs.

3. Those nodes that could not be executed before the initial node are added to a blacklist. The nodes in the blacklist are discarded because they are either a successor of the nodes in the slicing criterion (and thus they are executed always after it), or they are executed in a branch

of a choice that cannot lead to the slicing criterion. The blacklist is computed by iteratively collecting all the nodes that are a (control) successor of the nodes in the previous blacklist (initially the slicing criterion); and it also adds to the blacklist those nodes that are only synchronized with nodes in the blacklist.

4. A set of *pending* nodes that should be considered is computed. This set contains nodes that are synchronized with the nodes in *Meb* (thus they are executed at the same time). Therefore, synchronizations are followed in order to reach new nodes that must be executed before the slicing criterion. These steps are repeated until no new nodes are reached.

This algorithm always terminates.

The CEB analysis computes the set of nodes in the CSCFG that could be executed before a given node n. This means that all those nodes that must be executed before n are included, but also those nodes that are executed before n in some executions, and they are not in other executions (e.g., due to non-synchronized parallelism). Therefore, $MEB \subseteq CEB$.

The algorithm to compute the CEB analysis is presented in Paper 1. It, roughly, traverses the CSCFG forwards following all the paths that could be executed in parallel to nodes in *MEB*. In particular, the algorithm computes for each node in slicing criterion's nodes a set containing the part of the specification that could be executed before it. Then, it returns CEB as the union of all these sets. Each set *Ceb* is computed as follows:

1. First, the set *Ceb* is initialized with all those specification positions that must be executed before a node n because, trivially, they could be executed before it.

2. After, the set *loopnodes* is initialized. This set represents the nodes that belong to a loop in the computation executed before the slicing criterion was reached. For instance, in the process `A=(a→A)□(b→SKIP)` the left branch of the choice is a loop that could be executed several times before the slicing criterion, say `b`, was executed. Initially, this set contains the first node in a branch of a choice operator that does not belong to *Ceb* but can reach *Ceb* through a loop arc.

3. The set *loopnodes* is computed in the first loop of the algorithm and they are finally added to the slice (i.e., *Ceb*). The idea is to check that

the whole loop could be executed before the slicing criterion. If some sentence of the loop could not be executed before (e.g., because it is synchronized with an event that must occur after the slicing criterion), then the loop is discarded and not included in the slice.

4. A second loop of the algorithm is used to collect all those nodes that could be executed in parallel to the nodes in the slice (in *Ceb*). In particular, it traverses branches executed in parallel to nodes in *Ceb* until a node that could not be executed before the slicing criterion is found. For instance, consider the process $\texttt{A=(a}\rightarrow\texttt{b}\rightarrow\texttt{SKIP)}||_{\{\texttt{b}\}}\texttt{(c}\rightarrow\texttt{b}\rightarrow\texttt{SKIP)}$; and let us assume that the slicing criterion is `c`. Similarly to the first loop of the algorithm, the second loop traverses the left branch of the parallelism operator forwards until an event that could not be executed before the slicing criterion is found (in this example, `b`). Therefore, $\texttt{a}\rightarrow$ would be included in the slice.

The algorithms presented can extract a slice from any specification built from the considered CSP's syntax. However, note that only two operators have a special treatment in the algorithms: choices (because they introduce alternative computations) and synchronized parallelism constructs (because they introduce synchronization). Other operators such as prefixing, interleaving or sequential composition are only taken into account in the CSCFG construction phase; and they can be treated similarly in the algorithm (i.e., they are traversed forwards or backwards by the algorithm when exploring computations).

3.3 Implementation

We have implemented the MEB and CEB analyses and the algorithms to build the CSCFG for ProB. ProB [49] is an animator for the B-Method which also supports other languages such as CSP [16, 50]. ProB has been implemented in Prolog and it is publicly available [4].

Our tool is called SOC (which stands for *Slicing Of CSP*) and it is a development branch of ProB that is distributed and maintained for Mac, Linux and Windows. In SOC, the slicing process is completely automatic. Once the user has loaded a CSP specification, she can select (with the mouse) the event, operator or process call she is interested in. Obviously, this simple

3.3 Implementation

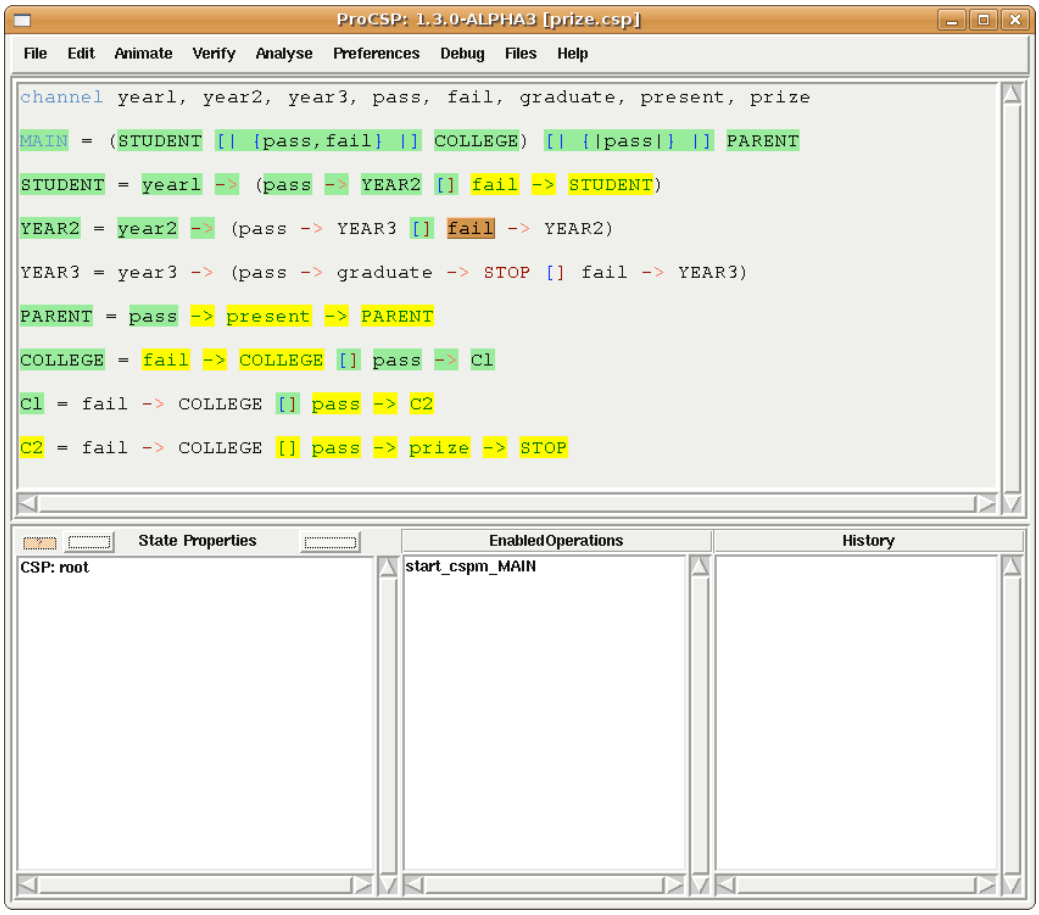


Figure 3.8: Slice of a CSP specification produced by SOC

action is enough to define a slicing criterion because the tool can automatically determine the process and the source position of interest. Then, the tool internally generates an internal data structure (the CSCFG) that represents all possible computations, and uses the MEB and CEB analyses to construct the slices. The result is shown to the user by highlighting the part of the specification that must (respectively could) be executed before the specified event. Figure 3.8 shows a screenshot of the tool showing a slice of the specification in Figure 3.1. SOC also includes a transformation to convert slices into executable programs. This allows us to use SOC for program specialization. The specialized versions can be directly executed in ProB.

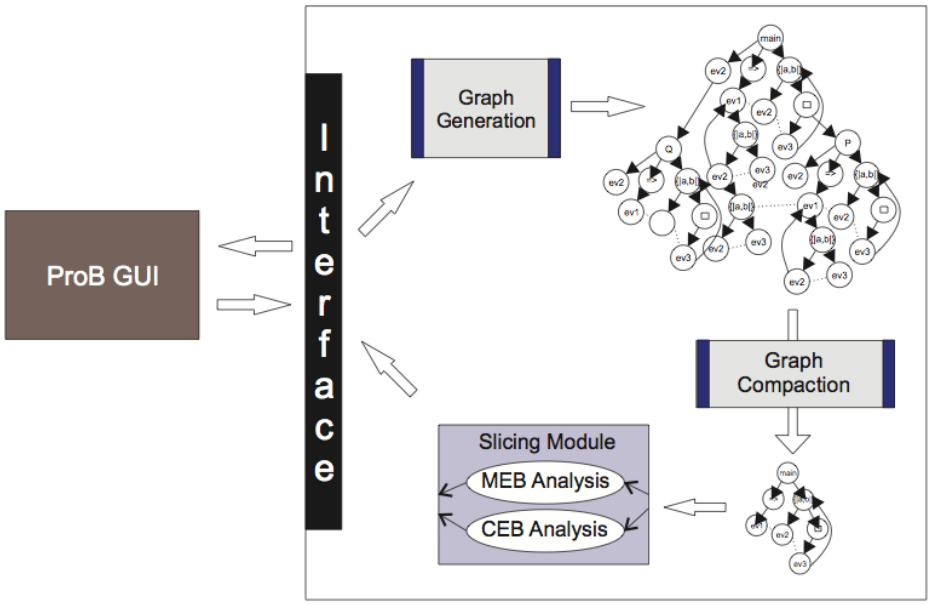


Figure 3.9: Slicer's Architecture

3.3.1 Architecture of SOC

SOC has been implemented in Prolog and it has been integrated in ProB. Therefore, SOC can take advantage of ProB's graphical features to show slices to the user. In order to be able to color parts of the code, it has been necessary to implement the source code positions detection in such a way that ProB can color every subexpression that is sliced away by SOC.

Figure 3.9 summarizes the internal architecture of SOC. Note that both the graph compaction module and the slicing module take a CSCFG as input, and hence, they are independent of CSP. Apart from the interface module for the communication with ProB, SOC has three main modules that we describe in the following:

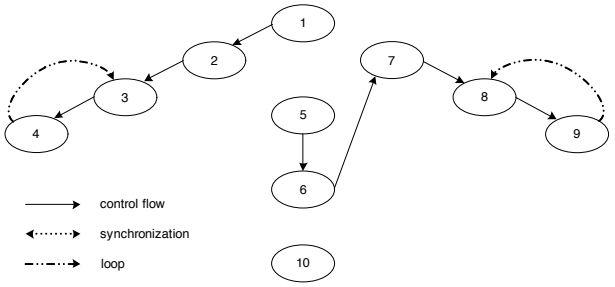
Graph Generation

The first task of the slicer is to build a CSCFG. The module that generates the CSCFG from the source program is the only module that is CSP dependent. This means that SOC could be used with other languages by only changing the graph generation module.

Nodes and arcs are built following the algorithm described in Paper 2. For efficiency reasons, the implementation of the CSCFG makes some simplifications that reduce the size of the graph. For instance, "*start*" and "*end*" nodes are not present in the graph. Another simplification to reduce the size of the graph is graph compaction (described below).

We have implemented two versions of this module. The first version aims to producing a precise analysis. For this purpose, the original notion of context is modified to introduce loop arcs in the graph whenever a specification position is repeated in a loop. However, this notion of context can produce big CSCFGs for some examples. This implies more memory usage and more time to compute the graphs and the slices. In such cases, the user could be interested in producing the CSCFG as fast as possible; for instance, when the analysis is used as a preprocessing stage of another analysis. Therefore, we have produced a lightweight version to produce a fast analysis when necessary. This second version uses a relaxed notion of context that allows the CSCFG to cut more branches of the graph with loop arcs. In the fast analysis, we skip the restriction that a specification position must be repeated. Therefore, while the *precise* context only introduces a loop arc in the CSCFG when the same specification position is repeated in a branch, the *fast* context introduces a loop arc when the same process call is repeated, even if the specification position of the call is different.

Consider again the CSCFG in Figure 3.7. This CSCFG corresponds to the precise context, and thus loop arcs are only used when the same specification position is repeated. In contrast, the CSCFG constructed using the fast context uses loop arcs whenever the same process call is repeated (i.e., the literal). It is depicted in Figure 3.10.



Node	Sequence of nodes
1	start (MAIN,0), P (MAIN,1), start (MAIN,1)
2	Q (P,Λ), start (P,Λ)
3	P (Q,Λ), start (Q,Λ)
4	Q (P,Λ)
5	end (Q,Λ), end (P,Λ), end (MAIN,1), ; (MAIN,Λ)
6	P (MAIN,2), start (MAIN,2)
7	Q (P,Λ), start (P,Λ)
8	P (Q,Λ), start (Q,Λ)
9	Q (P,Λ)
10	end (Q,Λ), end (P,Λ), end (MAIN,2), end (MAIN,0)

Figure 3.10: CSCFG of the specification in Figure 3.7 using the fast context.

Both analyses have been compared with several benchmarks. The results are presented in Section 3.3.2.

Graph Compaction

For the sake of clarity, the definition of CSCFG proposed so far does not take into account efficiency. In particular, it includes several nodes that are unnecessary from an implementation point of view. Therefore, we have implemented a module that reduces the size of the CSCFG by removing superfluous nodes and by joining together those nodes that form paths that the slicing algorithms must traverse in all cases. This compaction not only reduces the size of the stored CSCFG, but it also speeds up the slicing process due to the reduced number of nodes to be processed.

For instance, the graph of Figure 3.11 is the compacted version of the CSCFG in Figure 3.3(b). Here, e.g., node `2` accounts for the sequence of nodes `BUS` and `start BUS`. The compacted version is a very convenient representation because the reduced data structure speeds up the graph traversal process. In practice, the graph compaction phase reduces the size of the graph up to 40% on average.

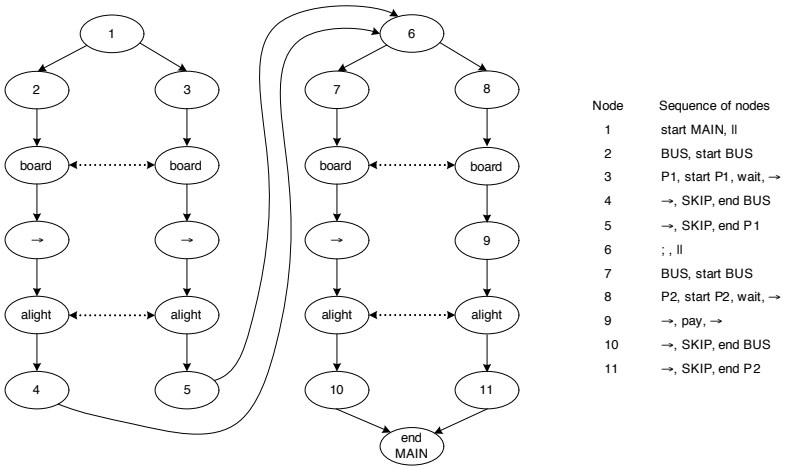


Figure 3.11: Compacted version of the CSCFG in Figure 3.3(b)

Slicing Module

This is the main module of the tool. It is further composed of two submodules that implement the algorithms to perform the MEB and CEB analyses on the compacted CSCFGs. This module extracts two subgraphs from the compacted CSCFG using both MEB and CEB. Then, it extracts from the

(a) Benchmark time results for the FAST CONTEXT

Benchmark	CSCFG	MEB	CEB	Total
ATM.csp	805 ms.	36 ms.	67 ms.	908 ms.
RobotControl.csp	277 ms.	39 ms.	21 ms.	337 ms.
Buses.csp	29 ms.	2 ms.	1 ms.	32 ms.
Prize.csp	55 ms.	35 ms.	10 ms.	100 ms.
Phils.csp	72 ms.	12 ms.	4 ms.	88 ms.
TrafficLights.csp	103 ms.	20 ms.	12 ms.	135 ms.
Processors.csp	10 ms.	4 ms.	2 ms.	16 ms.
ComplexSync.csp	212 ms.	264 ms.	38 ms.	514 ms.
Computers.csp	23 ms.	6 ms.	1 ms.	30 ms.
Highways.csp	11452 ms.	100 ms.	30 ms.	11582 ms.

(b) Benchmark time results for the PRECISE CONTEXT

Benchmark	CSCFG	MEB	CEB	Total
ATM.csp	10632 ms.	190 ms.	272 ms.	11094 ms.
RobotControl.csp	2603 ms.	413 ms.	169 ms.	3185 ms.
Buses.csp	25 ms.	1 ms.	0 ms.	26 ms.
Prize.csp	352 ms.	317 ms.	79 ms.	748 ms.
Phils.csp	96 ms.	12 ms.	8 ms.	116 ms.
TrafficLights.csp	2109 ms.	1678 ms.	416 ms.	4203 ms.
Processors.csp	15 ms.	2 ms.	5 ms.	22 ms.
ComplexSync.csp	23912 ms.	552 ms.	174 ms.	24638 ms.
Computers.csp	51 ms.	4 ms.	6 ms.	61 ms.
Highways.csp	58254 ms.	1846 ms.	2086 ms.	62186 ms.

Table 3.1: Benchmark time results for the FAST and PRECISE CONTEXT

subgraphs the part of the source code which forms the slice. This information can be extracted directly from the graph because its nodes are labeled with the specification positions to be highlighted. If the user has selected to produce an executable slice, then the slice is further transformed to become executable (it mainly fills gaps in the produced slice in order to respect the syntax of the language). The final result is then returned to ProB in such a way that ProB can either highlight the final slice or save a new CSP executable specification in a file.

3.3.2 Benchmarking the Slicer

In order to measure the performance and the slicing capabilities of our tool, we conducted some experiments over the following benchmarks:

(a) Benchmark size results for the FAST CONTEXT

Benchmark	Ori_CSCFG	Com_CSCFG	(%)	MEB Slice	CEB Slice
ATM.csp	156 nodes	99 nodes	63.46 %	32 nodes	45 nodes
RobotControl.csp	337 nodes	121 nodes	35.91 %	22 nodes	109 nodes
Buses.csp	20 nodes	20 nodes	90.91 %	11 nodes	11 nodes
Prize.csp	70 nodes	52 nodes	74.29 %	25 nodes	42 nodes
Phils.csp	181 nodes	57 nodes	31.49 %	9 nodes	39 nodes
TrafficLights.csp	113 nodes	79 nodes	69.91 %	7 nodes	60 nodes
Processors.csp	30 nodes	15 nodes	50.00 %	8 nodes	9 nodes
ComplexSync.csp	103 nodes	69 nodes	66.99 %	37 nodes	69 nodes
Computers.csp	53 nodes	34 nodes	64.15 %	18 nodes	29 nodes
Highways.csp	103 nodes	62 nodes	60.19 %	41 nodes	48 nodes

(b) Benchmark size results for the PRECISE CONTEXT

Benchmark	Ori_CSCFG	Com_CSCFG	(%)	MEB Slice	CEB Slice
ATM.csp	267 nodes	165 nodes	61.8 %	52 nodes	59 nodes
RobotControl.csp	1139 nodes	393 nodes	34.5 %	58 nodes	369 nodes
Buses.csp	22 nodes	20 nodes	90.91 %	11 nodes	11 nodes
Prize.csp	248 nodes	178 nodes	71.77 %	15 nodes	47 nodes
Phils.csp	251 nodes	56 nodes	22.31 %	9 nodes	39 nodes
TrafficLights.csp	434 nodes	267 nodes	61.52 %	7 nodes	217 nodes
Processors.csp	37 nodes	19 nodes	51.35 %	8 nodes	14 nodes
ComplexSync.csp	196 nodes	131 nodes	66.84 %	18 nodes	96 nodes
Computers.csp	109 nodes	72 nodes	66.06 %	16 nodes	67 nodes
Highways.csp	503 nodes	275 nodes	54.67 %	47 nodes	273 nodes

Table 3.2: Benchmark size results for the FAST and PRECISE CONTEXT

- `ATM.csp`. This specification represents an Automated Teller Machine. The slicing criterion is `(Menu,getmoney)`, i.e., we are interested in determining what parts of the specification must be executed before the menu option `getmoney` is chosen in the ATM.
- `RobotControl.csp`. This example describes a game in which four robots move in a maze. The slicing criterion is `(Referee,winner2)`, i.e., we want to know what parts of the system could be executed before the second robot wins.
- `Buses.csp`. This example describes a bus service with two buses running in parallel. The slicing criterion is `(BUS37, pay90)`, i.e., we are interested in determining what could and could not happen before the user payed at bus 37.
- `Prize.csp`. This is the specification of Figure 3.1. Here, the slicing criterion is `(YEAR2, fail)`, i.e., we are interested in determining what

parts of the specification must be executed before the student fails in the second year.

- `Phils.csp`. This is a simple version of the dining philosophers problem. In this example, the slicing criterion is `(PHIL221, DropFork2)`, i.e., we want to know what happened before the second philosopher dropped the second fork.

- `TrafficLights.csp`. This specification defines two cars driving in parallel on different streets with traffic lights for cars controlling. The slicing criterion is `(STREET3,park)`, i.e., we are interested in producing an executable version of the specification in which we could simulate the executions where the second car parks on the third street.

- `Processors.csp`. This example describes a system that, once connected, receives data from two different machines. The slicing criterion is `(MACH1,datreq)` to know what parts of the example must be executed before the first machine requests data.

- `ComplexSync.csp`. This specification defines five routers working in parallel. Router i can only send messages to router $i + 1$. Each router can send a broadcast message to all routers. The slicing criterion is `(Process3,keep)`, i.e., we want to know what parts of the system could be executed before router 3 keeps a message.

- `Computers.csp`. This benchmark describes a system in which a user can surf internet and download files. The computer can check wether files are infected by virus. The slicing criterion is `(USER,consult_file)`, i.e., we are interested in determining what parts of the specification must be executed before the user consults a file.

- `Highways.csp`. This specification describes a net of spanish highways. The slicing criterion is `(HW6,Toledo)`, i.e., we want to determine what cities must be traversed in order to reach Toledo from the starting point.

All the source code and other information about the benchmarks can be found at [51].

For each benchmark, Table 3.0(a) and Table 3.0(b) summarize the time spent to generate the compacted CSCFG (this includes the generation plus the compaction phases), to produce the MEB and CEB slices (since CEB analysis uses MEB analysis, CEB's time corresponds only to the time spent after

performing the MEB analysis), and the total time. Table 3.0(a) shows the results when using the fast context and Table 3.0(b) shows the results associated to the precise context. Clearly, the fast context achieves a significative time reduction. In these tables we can observe that Highways.csp needs more time even though the size of its associated CSCFG is similar to the other examples. Almost all the time needed to construct the CSCFG is used in computing the synchronizations. The high number of synchronizations performed in Highways.csp is the cause of its expensive cost.

Table 3.1(a) and Table 3.1(b) summarize the size of all objects participating in the slicing process for both the fast and the precise contexts respectively: Column `Ori_CSCFG` shows the size of the CSCFG of the original program. Observe that the precise context can increase the size of the CSCFG up to four times with respect to the fast context. Column `Com_CSCFG` shows the size of the compacted CSCFG. Column `(%)` shows the percentage of the compacted CSCFG' size with respect to the original CSCFG. Note that in some examples the reduction is almost 70% of the original size. Finally, columns `MEB Slice` and `CEB Slice` show respectively the size of the MEB and CEB CSCFG' slices. Clearly, CEB slices are always equal or greater than their MEB counterparts.

The CSCFG compaction technique seems to be useful. Experiments show that the size of the original specification is substantially reduced using this technique. The size of both MEB and CEB slices obviously depends on the slicing criterion selected. Table 3.1(a) and Table 3.1(b) compare both slices with respect to the same criterion but different contexts and, therefore, they give an idea of the difference between them.

SOC is open and publicly available. All the information related to the experiments, the source code of the benchmarks, the slicing criteria used, the source code of the tool and other material related to the project can be found at [51].

3.4 Related work

Program slicing has been already applied to concurrent programs of different programming paradigms, see e.g. [89, 90]. As a result, different graph representations have arisen to represent synchronization. The first proposal of a program slicing method for concurrent programs by Cheng [21] was later im-

proved by Krinke [45, 46] and Nanda [67]. All these approaches are based on the so called *threaded control flow graph* and the *threaded program dependence graph*. Unfortunately, their approaches are not appropriate for slicing CSP, because their work is based on a different kind of synchronization. They use the following concept of *interference* to represent program synchronization.

(Interference) A node $S1$ is *interference* dependent on a node $S2$ if $S2$ defines a variable v, $S1$ uses the variable v and $S1$ and $S2$ execute in parallel.

In CSP, in contrast, a synchronization happens between two processes if the synchronized event is executed at the same time by both processes. In addition, both processes cannot proceed in their executions until they have synchronized. This is the key point that underpin our MEB and CEB analyses. This idea has been already exploited in the *concurrent control flow graph* [33] which allows us to model the phenomenon known as *fully-blocking* semantics where a process sending a message to other process is blocked until the other receives the message and vice versa. This is equivalent to our synchronization model. In these graphs, as in previous approaches (and in conventional program slicing in general), the slicing criterion is a variable in a point of interest, and the slice is formed by the sentences that *influence* this variable due to control and data dependences. For instance, consider the following program fragment:

(1) $read(x)$;
(2) $print(x)$;
(3) $if \;\; x > 0$
(4) $\quad then \; y = x - 1$;
(5) $\quad else \; y = \; 42$;
(6) $print(y)$;
(7) $z = y$;

A slice with respect to $(7, z)$ would contain sentences (1), (3), (4) and (5); because z data depends on y, y data depends on x and (4) and (5) control depend on (3). Sentences (2) and (6) would be discarded because they are print statements and thus, they do not have an influence on z.

In contrast, in our technique, if we select (7) as the slicing criterion, we get sentences (1), (2), (3) and (6) as the MEB slice because these sentences must be executed before the slicing criterion in all executions. The CEB slice would contain the whole program.

Therefore, the purpose of our slicing technique is essentially different from previous work: while other approaches try to answer the question "*what parts of the program can influence the value of this variable at this point?*", our technique tries to answer the question "*what parts of the program must be executed before this point? and what parts of the program can be executed before this point?*". Therefore, our slicing criterion is different, but also the data structure we use for slicing is different. In contrast to previous work, we do not use a PDG like graph, and use instead a CFG like graph, because we focus on control flow rather than control and data dependence.

Despite the problem being undecidable, determining the MEB and CEB slices can be very useful and has many different applications such as debugging, program comprehension, program specialization and program simplification. Surprisingly, to the best of our knowledge, our approach is the first one to address the problem in a concurrent and explicitly synchronized context. In fact, the data structure most similar to the CSCFG is the SCFG by Callahan and Sublok [17] (see Section 3.1 for a description of this data structure, and a comparison with our CSCFG). Unfortunately, the SCFG does not take the calling context into account and thus it is not appropriate for the MEB and CEB analyses.

Our technique is not the first approach that applies program slicing to CSP specifications. Program slicing has also been applied to CSP by Bruckner and Wehrheim who introduced a method to slice CSP-OZ specifications [14]. Nevertheless, their approach ignores CSP synchronization and focus instead on the OZ's variables. As in previous approaches, their slicing criterion is an LTL formulae constructed with OZ's variables; and they use the standard PDG to compute the slice with a backwards reachability analysis.

Chapter 4

Tracking CSP Computations

In this chapter, we introduce the theoretical basis for tracking concurrent and explicitly synchronized computations in process algebras such as CSP. Tracking computations is a difficult task due to the subtleties of the underlying operational semantics which combines concurrency, non-determinism and non-termination. In CSP a trace is a sequence of events. Concretely, the operational semantics of CSP is an event-based semantics in which the occurrence of events fires the rules of the semantics. Hence, the final trace of the computation is the sequence of events occurred (see Chapter 8 of [81] for a detailed study of this kind of traces).

In this work we introduce an essentially different notion of trace [23] called track. In our setting, a track is a data structure which represents the sequence of expressions that have been evaluated during the computation, and moreover, this data structure is labelled with the location of these expressions in the specification. Therefore, a CSP track is much more informative than a CSP trace since the former not only contains a lot of information about original program structures but also explicitly relates the sequence of events with the parts of the specification that caused these events.

Consider the CSP specification shown in Figure 4.1. This specification models several gambling activities running in parallel and modelled by process `GAMBLING`. One of the games is the casino. A `CASINO` is modelled as the interaction of three parallel processes, namely a `PLAYER`, a `ROULETTE`, and a `CROUPIER`. The player bets for red, and she can win a prize or not. The roulette simply takes a color (either red or black); and the croupier checks the bet and the color of the roulette in order to give a prize to the player or

$$\texttt{MAIN} = \texttt{CASINO} \;||\; \texttt{GAMBLING}$$

$$\texttt{CASINO} = (\texttt{PLAYER} \;|||\; \texttt{ROULETTE}) \;\underset{\{\texttt{betred},\texttt{red},\texttt{black},\texttt{prize}\}}{\|}\; \texttt{CROUPIER}$$

$$\texttt{PLAYER} = \texttt{betred} \rightarrow (\texttt{prize} \rightarrow \texttt{STOP} \;\Box\; \texttt{noprize} \rightarrow \texttt{STOP})$$

$$\texttt{ROULETTE} = \texttt{red} \rightarrow \texttt{STOP} \;\Box\; \texttt{black} \rightarrow \texttt{STOP}$$

$$\begin{aligned}\texttt{CROUPIER} = {} & (\texttt{betred} \rightarrow \texttt{red} \rightarrow \texttt{prize} \rightarrow \texttt{STOP}) \\ & \Box\; (\texttt{betred} \rightarrow \texttt{black} \rightarrow \texttt{prize} \rightarrow \texttt{STOP}) \\ & \Box\; (\texttt{betblack} \rightarrow \texttt{black} \rightarrow \texttt{prize} \rightarrow \texttt{STOP}) \\ & \Box\; (\texttt{betblack} \rightarrow \texttt{red} \rightarrow \texttt{getmoney} \rightarrow \texttt{STOP})\end{aligned}$$

$$\texttt{GAMBLING} = \textit{Complex Composite Processes}$$

Figure 4.1: CSP specification of gambling activities

just get the bet money.

This specification contains an error, because it allows the trace of events $t = \langle \texttt{betred}, \texttt{black}, \texttt{prize} \rangle$ where the player bets for red and she wins a prize even though the roulette takes black.

Now assume that we execute the specification and discover the error after executing trace t. A track can be very useful to understand why the error was caused, and what part of the specification was involved in the wrong execution. For instance, if we look at the track of Fig. 4.2, we can easily see that the three processes run in parallel, and that the prize is given because there is a synchronization (dashed edges represent synchronizations) between CROUPIER and PLAYER that should never happen. Observe that the track is intuitive enough as to be a powerful program comprehension tool that provides much more information than the trace.

Moreover, observe that the track contains explicit information about the specification's expressions that were involved in the execution. Therefore, it can be used for program slicing (see [86, 84] for an explanation of the technique and Paper 1 for an adaptation of program slicing to CSP). In particular, in this example, we can use the track to extract the part of the

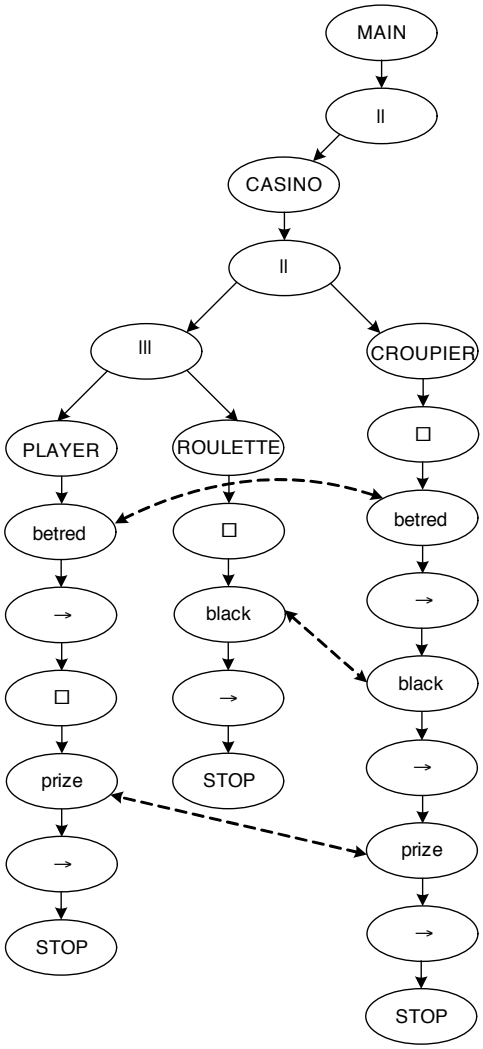


Figure 4.2: Track of the program in Figure 4.1

program that was involved in the execution—note that this is the only part that could cause the error—. This part has been underscored in the example. With a quick look, one can see that the underscored part of process `CROUPIER` produced the wrong behavior. Event `prize` should be replaced by `getmoney`.

Another interesting application of tracks is related to component extraction and reuse. If we are interested in a particular trace, and we want to extract the part of the specification that models this trace to be used in another model, we can simply produce a slice, and slightly augment the code to make it syntactically correct (see Paper 1 for an example and an explanation

of this transformation). In our example, even though the system is very big due to process `GAMBLING`, the track is able to extract the only information related to the trace.

In Paper 3 we define an instrumented operational semantics that generates as a side-effect an appropriate data structure (a track) that can be used to track computations. Formal definition of a tracking semantics improves the understanding of the tracking process, but also, it allows us to formally prove the correctness of the computed tracks.

Additionally, we have implemented the first tool [55] able to produce tracks automatically.

4.1 Tracking Computations

A track is formed by the sequence of expressions that are evaluated during an execution. These expressions are conveniently connected to form a graph. However, several program analysis techniques such as program slicing make use of the locations of program expressions, and thus, this notion of track is insufficient for them. Therefore, we want our tracks to also store the location of each literal (i.e., events, operators and process names) in the specification so that the track can be used to know what portions of the source code have been executed and in what order. The inclusion of source positions in the track implies an additional level of complexity in the semantics, but the benefits of providing our tracks with this additional information are clear and, for some applications, essential. Therefore, we use labels (that we call *specification positions*) to uniquely identify each literal in a specification which roughly corresponds to nodes in the CSP specification's abstract syntax tree. Consider, for example, the CSP specification of Figure 4.3 where literals are labelled with their associated specification positions (they are underlined) so that labels are unique.

In order to introduce the definition of track, we need first to define the concept of *control-flow*, which refers to the order in which the individual literals of a CSP specification are executed. Intuitively, the control can pass from a specification position α to a specification position β iff an execution exists where α is executed before β. This notion of control-flow is similar to the control-flow used in the *control-flow graphs* (CFG) [86] of imperative programming. We have adapted the same idea to CSP where choices and

$$\texttt{MAIN}_{\underline{(\texttt{MAIN},0)}} = (\texttt{a}_{\underline{(\texttt{MAIN},1.1)}} \rightarrow_{\underline{(\texttt{MAIN},1)}} \texttt{STOP}_{\underline{(\texttt{MAIN},1.2)}}) \underset{\{\texttt{a}\}}{\parallel}_{\underline{(\texttt{MAIN},\Lambda)}} (\texttt{P}_{\underline{(\texttt{MAIN},2.1)}} \Box_{\underline{(\texttt{MAIN},2)}} (\texttt{a}_{\underline{(\texttt{MAIN},2.2.1)}} \rightarrow_{\underline{(\texttt{MAIN},2.2)}} \texttt{STOP}_{\underline{(\texttt{MAIN},2.2.2)}}))$$

$$\texttt{P}_{\underline{(\texttt{P},0)}} = \texttt{b}_{\underline{(\texttt{P},1)}} \rightarrow_{\underline{(\texttt{P},\Lambda)}} \texttt{SKIP}_{\underline{(\texttt{P},2)}}$$

Figure 4.3: Labelled CSP specification

parallel composition appear; and in a similar way to the CFG, we use this definition to draw control arcs in our tracks. Indeed, this notion of control-flow is the same used in Chapter 3 to build the CSCFG.

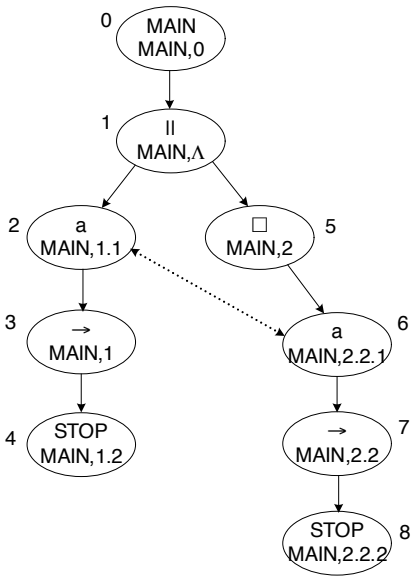
For instance, in the specification of Figure 4.3, we can see how the control can pass from a specification position to another one, e.g., we have $(\texttt{MAIN}, 2) \Rightarrow (\texttt{MAIN}, 2.1)$ and $(\texttt{MAIN}, 2) \Rightarrow (\texttt{MAIN}, 2.2.1)$. And $(\texttt{MAIN}, 2.2.1) \Rightarrow (\texttt{MAIN}, 2.2)$; $(\texttt{MAIN}, 2.2) \Rightarrow (\texttt{MAIN}, 2.2.2)$ and $(\texttt{MAIN}, 2.1) \Rightarrow (\texttt{P}, 1)$.

Control-flow is defined statically and says whether the control can pass from α to β in some derivation. However, the track is a dynamic structure produced for a particular derivation. Therefore, we produce a dynamic version of the definition of control-flow which is defined for a particular derivation. For example, consider again the specification of Figure 4.3. We show in Fig. 4.4(a) one possible derivation (ignoring subderivations) of this specification (for the time being, the underlined part should be ignored). Its associated track is shown in Fig. 4.4(b). In the example, we see that the track is a connected and directed graph. Apart from the control-flow edges, there is one synchronization edge between nodes $(\texttt{MAIN}, 1.1)$ and $(\texttt{MAIN}, 2.2.1)$ representing the synchronization of event $\texttt{a}$.

The trace associated with the derivation in Fig. 4.4(a) is $\langle \texttt{a} \rangle$. Therefore, note that the track is much more informative: it shows the exact processes that have been evaluated with an explicit causality relation; and, in addition, it shows the specification positions that have been evaluated and in what order.

$$
\begin{aligned}
\texttt{MAIN} &\overset{\text{(PC)}}{\rightsquigarrow} (\texttt{a} \rightarrow \texttt{STOP}) \underset{\{\texttt{a}\}}{\parallel} (\texttt{P} \Box (\texttt{a} \rightarrow \texttt{STOP})) \\
&\overset{\text{(SP3)}}{\rightsquigarrow} \texttt{STOP} \underset{\{\texttt{a}\}}{\parallel} \texttt{STOP} \\
&\overset{\text{(SP1)}}{\rightsquigarrow} \bot \underset{\{\texttt{a}\}}{\parallel} \texttt{STOP} \\
&\overset{\text{(SP2)}}{\rightsquigarrow} \bot \underset{\{\texttt{a}\}}{\parallel} \bot
\end{aligned}
$$

(a) Derivation



(b) Track

Figure 4.4: Derivation and track associated with the specification of Fig. 4.3

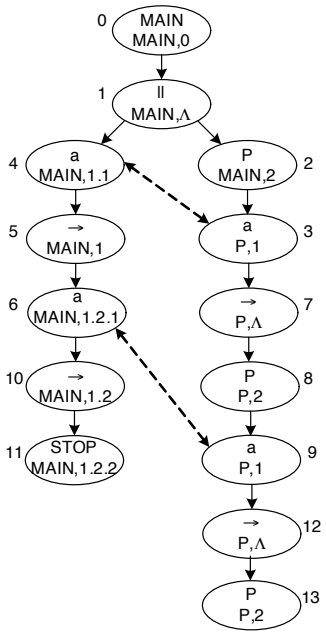
4.2 Instrumenting the Semantics for Tracking

The generation of tracks in CSP is a task that must overcome challenges such as non-deterministic execution of processes, deadlocks, non-terminating processes and synchronizations. In this work, we design a solution that success-

$$\texttt{MAIN} \rightarrow \underline{(\texttt{MAIN},0)} = \texttt{a}\underline{(\texttt{MAIN},1.1)} \rightarrow \underline{(\texttt{MAIN},1)}\texttt{a}\underline{(\texttt{MAIN},1.2.1)} \rightarrow \underline{(\texttt{MAIN},1.2)}\texttt{STOP}\underline{(\texttt{MAIN},1.2.2)}$$
$$\underset{\{\texttt{a}\}}{\|}\ \underline{(\texttt{MAIN},\Lambda)}\texttt{P}\underline{(\texttt{MAIN},2)}$$

$$\texttt{P}\underline{(\texttt{P},0)} = \texttt{a}\underline{(\texttt{P},1)} \rightarrow \underline{(\texttt{P},\Lambda)}\texttt{P}\underline{(\texttt{P},2)}$$

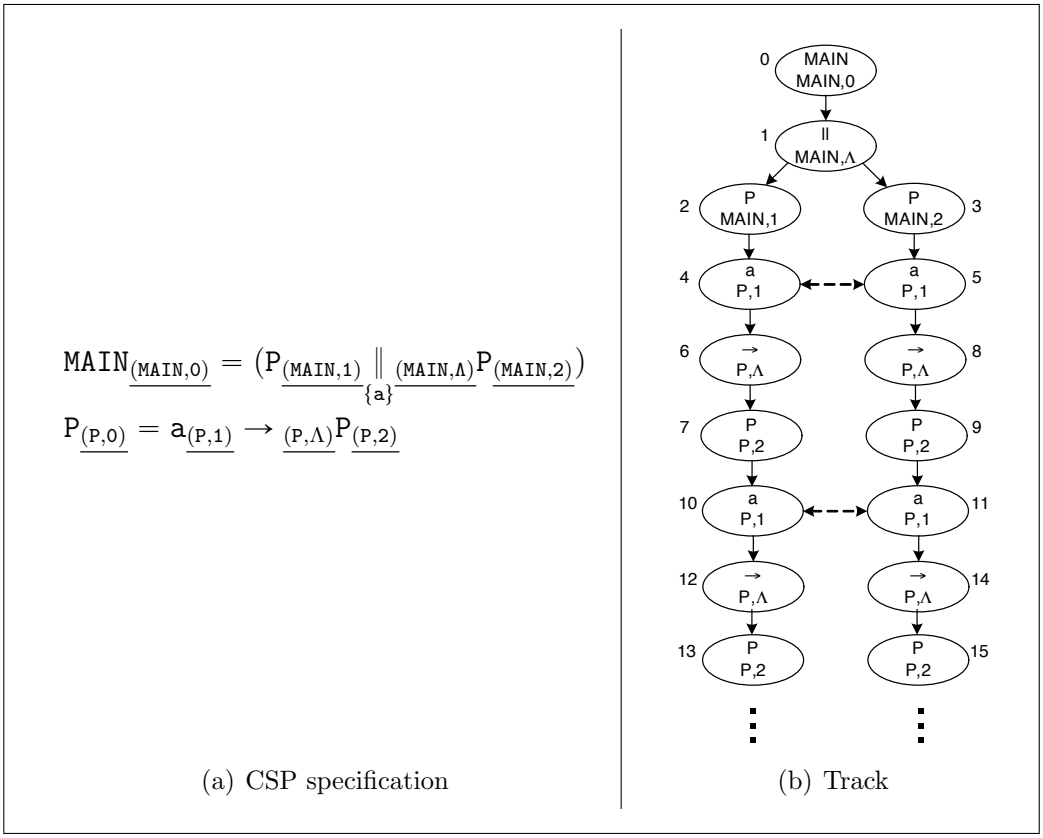
(a) CSP specification



(b) Track

Figure 4.5: CSP specification with two synchronized processes and its corresponding track

fully faces these difficulties. Firstly, we generate tracks with an augmented semantics that is conservative with respect to the standard operational semantics. Therefore, the execution order is the standard order, thus nondeterminism and synchronizations are solved by the semantics. Moreover, the semantics generates the track incrementally, step by step. Therefore, infinite computations can be tracked until they are stopped. Hence, it is not needed to actually finish a computation to get the track of the subcompu-



(a) CSP specification (b) Track

Figure 4.6: CSP specification of two infinite processes mutually synchronized and its corresponding track

tations performed. For example, consider the CSP specification of Figure 4.5(a) where each (sub)process has been labelled with its associated specification position (underlining). The final track computed is the graph of Fig. 4.5(b). Another interesting example is illustrated by the CSP specification of Figure 4.6(a) where two non-terminating processes run in parallel and synchronize infinitely. Because the computation is infinite, the track (shown in Fig. 4.6(b)) is also infinite.

In order to solve the problem of deadlocks (that stop the computation), and have a representation for them in the tracks, when a deadlock happens, the semantics performs some additional steps to be able to generate a part of the track that represents the deadlock. These additional steps do not influence the other rules of the semantics, thus it remains conservative.

(Process Call)
$$\frac{}{(N_\alpha, G, m, \Delta) \xrightarrow{\tau} (rhs(N), G[m \underset{n}{\mapsto} \alpha], n, \varnothing)}$$

(Prefixing)
$$\frac{}{(a_\alpha \rightarrow_\beta P, G, m, \Delta) \xrightarrow{a} (P, G[m \underset{n}{\mapsto} \alpha, n \underset{p}{\mapsto} \beta], p, \{m\})}$$

(SKIP)
$$\frac{}{(\texttt{SKIP}_\alpha, G, m, \Delta) \xrightarrow{\checkmark} (\Omega, G[m \underset{n}{\mapsto} \alpha], n, \varnothing)}$$

(STOP)
$$\frac{}{(\texttt{STOP}_\alpha, G, m, \Delta) \xrightarrow{\tau} (\bot, G[m \underset{n}{\mapsto} \alpha], n, \varnothing)}$$

(Internal Choice 1)
$$\frac{}{(P \sqcap_\alpha Q, G, m, \Delta) \xrightarrow{\tau} (P, G[m \underset{n}{\mapsto} \alpha], n, \varnothing)}$$

(Internal Choice 2)
$$\frac{}{(P \sqcap_\alpha Q, G, m, \Delta) \xrightarrow{\tau} (Q, G[m \underset{n}{\mapsto} \alpha], n, \varnothing)}$$

(External Choice 1)
$$\frac{(P_1, G', n', \Delta) \xrightarrow{\tau} (P', G'', n'', \varnothing)}{(P_1 \ \square_{(\alpha,n_1,n_2)} P_2, G, m, \Delta) \xrightarrow{\tau} (P' \ \square_{(\alpha,n'',n_2)} P_2, G'', m, \varnothing)}$$
where $(G', n') = \texttt{FirstEval}(G, n_1, m, \alpha)$

(External Choice 2)
$$\frac{(P_2, G', n', \Delta) \xrightarrow{\tau} (P', G'', n'', \varnothing)}{(P_1 \ \square_{(\alpha,n_1,n_2)} P_2, G, m, \Delta) \xrightarrow{\tau} (P_1 \ \square_{(\alpha,n_1,n'')} P', G'', m, \varnothing)}$$
where $(G', n') = \texttt{FirstEval}(G, n_2, m, \alpha)$

(External Choice 3)
$$\frac{(P_1, G', n', \Delta) \xrightarrow{e} (P', G'', n'', \Delta')}{(P_1 \ \square_{(\alpha,n_1,n_2)} P_2, G, m, \Delta) \xrightarrow{e} (P', G'', n'', \Delta')} \quad e \in \Sigma \cup \{\checkmark\}$$
where $(G', n') = \texttt{FirstEval}(G, n_1, m, \alpha)$

(External Choice 4)
$$\frac{(P_2, G', n', \Delta) \xrightarrow{e} (P', G'', n'', \Delta')}{(P_1 \ \square_{(\alpha,n_1,n_2)} P_2, G, m, \Delta) \xrightarrow{e} (P', G'', n'', \Delta')} \quad e \in \Sigma \cup \{\checkmark\}$$
where $(G', n') = \texttt{FirstEval}(G, n_2, m, \alpha)$

(Sequential Composition 1)
$$\frac{(P, G, m, \Delta) \xrightarrow{e} (P', G', m', \Delta')}{(P; Q, G, m, \Delta) \xrightarrow{e} (P'; Q, G', m', \Delta')} \quad e \in \Sigma \cup \{\tau\}$$

(Sequential Composition 2)
$$\frac{(P, G, m, \Delta) \xrightarrow{\checkmark} (\Omega, G', n, \varnothing)}{(P;_\alpha Q, G, m, \Delta) \xrightarrow{\tau} (Q, G'[n \underset{p}{\mapsto} \alpha], p, \varnothing)}$$

(Synchronized Parallelism 1)
$$\frac{(P_1, G', n', \Delta) \xrightarrow{e'} (P', G'', n'', \Delta')}{(P_1 \underset{X}{\parallel}_{(\alpha,n_1,n_2)} P_2, G, m, \Delta) \xrightarrow{e} (P' \underset{X}{\parallel}_{(\alpha,n'',n_2)} P_2, G'', m, \Delta')} \quad \begin{array}{l}(e = e' = a \ \wedge \ a \notin X) \\ \vee \ (e = \tau \ \wedge \ e' \in \{\tau, \checkmark\})\end{array}$$
where $(G', n') = \texttt{FirstEval}(G, n_1, m, \alpha)$

(Synchronized Parallelism 2)
$$\frac{(P_2, G', n', \Delta) \xrightarrow{e'} (P', G'', n'', \Delta')}{(P_1 \underset{X}{\parallel}_{(\alpha,n_1,n_2)} P_2, G, m, \Delta) \xrightarrow{e} (P_1 \underset{X}{\parallel}_{(\alpha,n_1,n'')} P', G'', m, \Delta')} \quad \begin{array}{l}(e = e' = a \ \wedge \ a \notin X) \\ \vee \ (e = \tau \ \wedge \ e' \in \{\tau, \checkmark\})\end{array}$$
where $(G', n') = \texttt{FirstEval}(G, n_2, m, \alpha)$

(Synchronized Parallelism 3)
$$\frac{RewritingStep_1 \qquad\qquad RewritingStep_2}{(P_1 \underset{X}{\parallel}_{(\alpha,n_1,n_2)} P_2, G, m, \Delta) \xrightarrow{a} (P_1' \underset{X}{\parallel}_{(\alpha,n_1'',n_2'')} P_2', G'', m, \Delta_1 \cup \Delta_2)} \quad a \in X$$
where $G'' = G_1'' \cup G_2'' \cup \{s_1 \overset{a}{\leftrightarrow} s_2 \mid s_1 \in \Delta_1 \wedge s_2 \in \Delta_2\}$
$\wedge \ RewritingStep_1 = (P_1, G_1', n_1', \Delta) \xrightarrow{a} (P_1', G_1'', n_1'', \Delta_1)$
$\wedge \ (G_1', n_1') = \texttt{FirstEval}(G, n_1, m, \alpha)$
$\wedge \ RewritingStep_2 = (P_2, G_2', n_2', \Delta) \xrightarrow{a} (P_2', G_2'', n_2'', \Delta_2)$
$\wedge \ (G_2', n_2') = \texttt{FirstEval}(G, n_2, m, \alpha)$

(Synchronized Parallelism 4)
$$\frac{}{(\Omega \underset{X}{\parallel}_{(\alpha,n_1,n_2)} \Omega, G, m, \Delta) \xrightarrow{\checkmark} (\Omega, G', r, \varnothing)}$$
where $G' = G[\{p \underset{r}{\mapsto} \mid p \underset{q}{\mapsto} \ \in G \text{ where } q \in \{n_1, n_2\}\}]$

Figure 4.7: An instrumented operational semantics to generate CSP tracks

This section introduces an instrumented operational semantics of CSP that generates as a side-effect the tracks associated with the computations performed with the semantics. The tracking semantics is introduced in Figure 4.7, where we assume that every literal in the program has been labelled with its specification position (denoted by a subscript, e.g., P_α). In the semantics, a *state* is a tuple (P, G, m, Δ), where P is the process to be evaluated (the *control*), G is a directed graph (i.e., the track built so far), m is a numeric reference to the current node in G, and Δ is a set of references to nodes that may be synchronized. Concretely, m references the node in G where the specification position of the control P must be stored. Reference m is a fresh[1] reference generated to add new nodes to G. The basic idea of the graph construction is to record the current control with the current reference in every step by connecting it to its parent. Every time an external event happens during the computation, this event is stored in the set Δ of the current state. Therefore, when a synchronized parallelism is evaluated, all the events that must be synchronized are in Δ. We use the special symbol $\perp$ to denote any process that is deadlocked. In order to perform computations, we construct an initial state (e.g., $(\texttt{MAIN}, \varnothing, 0, \varnothing)$) and (non-deterministically) apply the rules of the instrumented semantics. When the execution has finished or has been interrupted, the semantics has produced the track of the computation performed so far.

In order to give a general idea of how the semantics works, a brief explanation for the most relevant rules of the semantics follows:

(External Choice) This operator can develop both branches while τ events happen, until an external event or ✓ occurs. This means that the semantics can add nodes to both branches of the track alternatively, and thus, it needs to store the next reference to use in every branch of the choice. This is done by labelling choice operators with a tuple of the form (α, n_1, n_2) where α is the specification position of the choice operator; and n_1 and n_2 are respectively the references to be used in the left and right branches of the choice, and they are initialized to $\bullet$, a symbol used to express that the branch has not been evaluated yet.

(Sequential Composition) Two rules correspond the operation $P; Q$. The first one is used to evolve process P until it is finished. P is evolved to a new process which is put into the control. When P successfully finishes (it becomes Ω), ✓ happens. Then, the second rule is used and Q is put into the control. The sequential composition operator ; is added to the graph with

[1] We assume that fresh references are numeric and generated incrementally.

successor the reference to be used in the first node added in the subderivation associated with Q.

(Synchronized Parallelism) In a synchronized parallel composition, both parallel processes can be intertwiningly executed until a synchronized event is found. Therefore, nodes from both processes can be added interwoven to the graph. Hence, each parallelism operator is labelled with a tuple of the form (α, n_1, n_2) as it happens with external choices.

Four rules are used for this operator. The first two rules develop the branches of the parallelism until they are finished or until they must synchronize. The third rule is used to synchronize the parallel processes. In this case, both branches must perform a rewriting step with the same visible (and synchronized) event. Each branch derivation has a non-empty set of events (Δ_1, Δ_2) to be synchronized (note that this is a set because many parallelisms could be nested). Then, all references in the sets Δ_1 and Δ_2 are mutually linked with synchronization edges. Both sets are joined to form the new set of synchronized events. Finally, the fourth rule is used when none of the parallel processes can proceed because they already successfully finished. In this case, the control becomes Ω indicating the successful termination of the synchronized parallelism. This rule also adds to the graph the arcs from all the parents of the last references of each branch to the fresh reference in the new state. Note that the fact of generating the next reference in each rule allows this rule to connect the final node of both branches to the next node. This simplifies other rules such as (Sequential Composition) that already has the reference of the node ready.

We illustrate the semantics with a simple example. Consider again the specification in Example 4.3. Figure 4.4(a) shows one possible derivation (excluding subderivations) for this example. Note that the underlined part corresponds to the additional rewriting steps performed by the tracking semantics. This derivation corresponds to the execution of the instrumented semantics with the initial state $(\texttt{MAIN}, \varnothing, 0, \varnothing)$. This computation corresponds to the execution of the right branch of the choice (i.e., $\texttt{a} \rightarrow \texttt{STOP}$). The final state is $(\bot \underset{\{\texttt{a}\}}{\parallel}_{((\texttt{MAIN},\Lambda),9,10)} \bot, G', 1, \varnothing)$. The final track G' computed for this execution is depicted in Fig. 4.4(b) where we can see that nodes are numbered with the references generated by the instrumented semantics. Note that nodes 9 and 10 were prepared by the semantics (edges to them were produced) but never used because the subcomputations were stopped in STOP. Note also that the track contains all the parts of the specification executed by the semantics. This means that if the left branch of the choice had been developed (i.e.,

unfolding the call to P), this branch would also belong to the track.

We prove the correctness of the tracking semantics by showing that (i) the computations performed by the tracking semantics are equivalent to the computations performed by the standard semantics; and (ii) the graph produced by the tracking semantics is the track of the derivation. We also prove that the trace of a derivation can be automatically extracted from the track of this derivation. Consequently, two theorems are defined. The first theorem shows that the computations performed with the tracking semantics are all and only the computations performed with the standard semantics. The only difference between them from an operational point of view is that the tracking semantics needs to perform one step when a STOP is evaluated (to add its specification position to the track) and then finishes, while the standard semantics finishes without performing any additional step. The second theorem states the correctness of the tracking semantics by ensuring that the graph produced is the track of the computation.

4.3 Implementation

We have developed a tool called CSP-Tracker that implements a CSP interpreter with a tracker. The interpreter executes a CSP specification and simultaneously produces the track associated to the performed derivation. CSP-Tracker incorporates mechanisms to produce colored graphs that represent the tracks in a very intuitive way. CSP-Tracker implements the instrumented semantics in Fig. **??**, and thus it can generate the track of a (partial) derivation until it finishes or is stopped. This is specially useful for the analysis of non-termination. In CSP-Tracker, the tracking process is completely automatic. Once the user has loaded a CSP specification, she can (automatically) produce a derivation and the tool internally generates the associated track. Both the track and the trace can be stored in a file, or displayed in the screen by generating *Graphviz*[2] graphs. Figure 4.8 shows a screenshot of a interface of the tool showing the track and the trace of the specification in Example 4.1.

[2] http://www.graphviz.org/

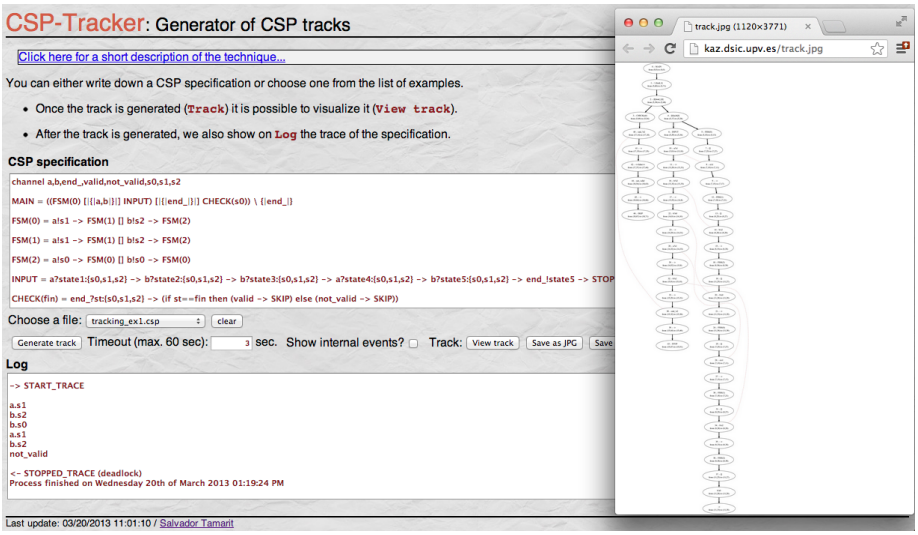


Figure 4.8: Track of a CSP specification produced by CSP-Tracker

4.3.1 Architecture of CSP-Tracker

The information collected by CSP-tracker is *dynamic*, and thus the subsequent analyses performed are very precise. We would like to allow the user to combine this information with other analyses that already exist for CSP. Therefore, we have integrated CSP-Tracker with the tool SOC (described in Chapter 3) that is able to perform different *static* analyses such as static slicing. Both tools are complementary and together can provide useful information, e.g., in debugging and program comprehension.

While SOC was implemented in Prolog, CSP-Tracker has been implemented in Erlang[3]. The election of Erlang was very conscious because Erlang is one of the most efficient languages for the use of multiple threads and parallelism; and it provides concurrent capabilities that enhance the execution of CSP specifications with the use of efficient message passing. In particular, with Erlang we can use truly concurrent processes to implement interleaving and synchronized parallelism.

All modules except the parser and the graph generator were implemented in Erlang. The CSP parser translates CSP to a Prolog representation that can be used by SOC. This parser is part of ProB [49, 50] which is one of the most extended IDE for CSP. Once CSP is translated to Prolog, we use a Prolog module to translate the resulting code to an Erlang structure. Note that this

[3] http://www.erlang.org/

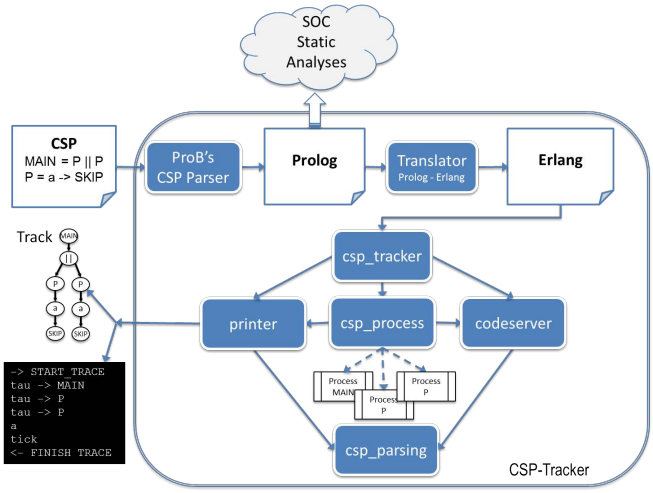


Figure 4.9: CSP-Tracker's Architecture

last step is somehow easy, because the syntax of Erlang was initially based on Prolog, so there are many similarities between them. In order to be able to graphically show tracks, we use Graphviz.

Figure 4.9 summarizes the internal architecture of CSP-Tracker. In the figure, the dark rectangles represent modules that are described in the following:

- `ProB's CSP parser`: It translates a CSP specification into a Prolog representation. This Prolog structure acts as an intermediate language that is used by SOC to perform complementary static analyses. This module is in charge of assigning specification positions. Observe in Fig. 4.10 that, for the sake of clarity, the tool uses lines and columns to identify literals. For instance, node 5 with literal `b` has the specification position `from (4,10) to (4,11)`; which means that `b` appears in the source code between columns 10 and 11 of line 4.

- `Translator Prolog-Erlang`: It produces an Erlang representation equivalent to the Prolog structure.

- `csp_tracker`: This module initializes the other modules. First, it loads the Erlang code produced and then it creates all the Erlang processes needed by the tool. Finally, it starts the execution of CSP process `MAIN`.

- `codeserver`: This module specifies a process that runs uninterruptedly during the generation of the track. It behaves as a server that stores

all the information about the code of the CSP processes. It waits for requests and serves them. A request is in fact a message that contains a process call. Then, `codeserver` returns a message containing the right hand side of the called process with the parameters substituted by the actual values of the arguments in the call.

- `printer`: This module also specifies a process that runs uninterruptedly and acts as a server. In this case, the requests contain information that should be used to print the trace of the execution or to generate the part of the track that represents the ongoing execution.

- `csp_process`: This module creates one Erlang process for each CSP process in the specification. All created processes run in parallel and synchronize via message passing when needed. Each of these processes interacts with `codeserver` and `printer` to perform process calls and generate the graph when required. For instance, the execution of a prefixing `a` $\rightarrow$ `P` calls `printer` to print `a` in the shell, and then calls `codeserver` to create a new process that represents `P`.

- `csp_parsing`: This module is basically a library with common functionality for the other modules.

4.3.2 Using CSP-Tracker

CSP-Tracker is publicly available including its source code as a GitHub repository [55]. There is also a web interface useful to test the tool. It can be found at [56].

This section shows the use of CSP-Tracker's web interface with three illustrative scenarios. The first scenario shows a specification that successfully finishes, the second scenario shows the case where the program is deadlocked and the third shows the case where the program produces an infinite computation.

In the first scenario, we consider a simple modification of Example 2.1.1, where all `STOP` terms have been replaced by `SKIP`. This change makes the process to finish successfully. We just load the first example and press the button `Generate Track`:

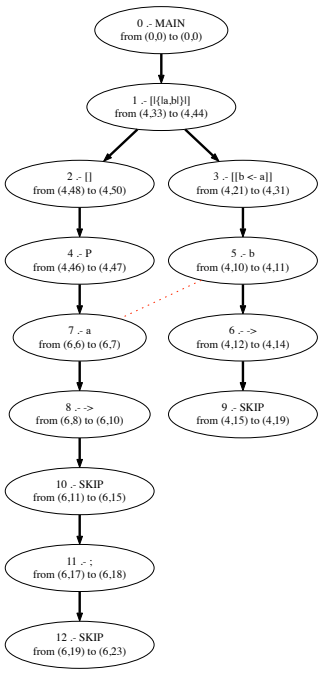


Figure 4.10: A track generated by CSP-Tracker

```
Creating the Erlang representation of the CSP file...
...
Created.

-> START_TRACE

   tau -> Call to process MAIN
   tau -> Call to process P
a
   tau
   tau
   tick

<- FINISH_TRACE
```

Once the execution is finished, a file `track.pdf` is produced containing the track associated to the execution of the CSP specification. The track generated by CSP-Tracker for this example is shown in Fig. 4.10.

During the execution, the trace produced is shown in the `Log` window. In the previous case, this trace was formed by both the internal and the external events fired by the semantics. This is interesting for a programmer that wants to analyse the behavior of CSP from a semantic point of view. However, the conventional programmer is only interested in the usual trace, only formed by external events. This can be obtained by unchecking the option `Show`

```
internal events?:

-> START_TRACE

a

<- FINISH_TRACE
```

Our second scenario is Example 4.1, that produces a deadlock. In this case, the tool automatically detects the deadlock and produces the trace until the deadlock happens. We have the following trace:

```
-> START_TRACE

betred
black
prize

<- STOPPED_TRACE (deadlock)
```

Our third scenario is Example 4.6(a) that produces an infinite loop. However, this is not a problem for the tracker, that can still run the specification and generate the track until it is stopped or a timeout is reached. We have the following trace:

```
-> START_TRACE

a
...
a

Timeout.
```

4.4 Related Work

In languages such as Haskell, the tracks (see, e.g., [23, 25, 24, 11]) are the basis of many analysis methods and tools. However, computing CSP tracks is a complex task due to the non-deterministic execution of processes, deadlocks, non-terminating processes and synchronizations. This is probably the reason why no correctness result exists that formally relates the track of a specification to its execution. This semantics is needed because it would allow us to prove important properties (such as correctness and completeness) of the techniques and tools based on tracking.

To the best of our knowledge, there is only one attempt to define and build tracks for CSP [13]. Their notion of track is based on the standard *program*

dependence graph [30]; therefore it is useful for program slicing but it is insufficient for other analyses that need a *context-sensitive graph* [46] (i.e., each different process call has a different representation). Moreover, their notion of track does not include synchronizations. Our tracks are able to represent synchronizations, and they are context-sensitive.

Chapter 5

From CSP Specifications to Petri Nets

This chapter introduces a new technique that allows us to automatically transform a CSP specification into an equivalent Petri net. The transformation is formally defined by instrumenting the operational semantics of CSP. Because the technique uses a semantics-directed transformation, it produces Petri nets that are closer to the CSP specification and thus easier to understand. This result is interesting because it allows CSP developers not only to graphically animate their specifications through the use of the equivalent Petri net, but it also allows them to use all the tools and analysis techniques developed for Petri nets. For these reasons, attempts to combine both models exist (see, e.g., [7]).

Transforming CSP to Petri nets is known to be useful since almost their origins, because it not only has a clear practical utility, but it also has a wide theoretical interest because both concurrent models are very different, and establishing relations between them allows us to extend results from one model to the other. In fact, the problem of transforming a CSP specification into an equivalent Petri net is complex due to the big differences that exist between both formalisms. For this reason, some previous approaches aiming to transform CSP to Petri nets have been criticized because, even though they are proved equivalent, it is hardly possible to see a relation between the generated Petri net and the initial CSP specification (i.e., when a transition of the Petri net is fired, it is not even clear to what CSP process corresponds this transition). In this respect, the transformation presented here is particularly interesting because the Petri net is generated directly from the operational

semantics in such a way that each syntactic element of the CSP specification has a representation in the Petri net. And, moreover, the sequences of steps performed by the CSP semantics are directly represented in the Petri net. Hence, it is not difficult to map the animation of the Petri net to the CSP specification.

Our transformation is based on an instrumentation of the CSP's operational semantics. Roughly speaking, we define an algorithm that explores all computations of a CSP specification by using the instrumented semantics. The execution of the semantics produces as a side-effect the Petri net associated with each computation, and thus the final Petri net is produced incrementally. Additionally, we state the correctness of the transformation from CSP to Petri nets. In particular, we prove that given a CSP specification, the exploring algorithm produces in finite time an equivalent Petri net.

5.1 Equivalence between CSP and Petri Nets

In order to formally prove the correctness of the transformation, we need a notion of equivalence that, in our case, is based on the traces generated by the initial CSP and the language produced by the final Petri net. In particular, given a CSP specification, the Petri net generated by our algorithm is equivalent to the CSP in the sense that the sequences of observable events produced are exactly the same in both models (i.e., they are equivalent modulo a given alphabet). In CSP terminology, these sequences are the so-called traces (see, e.g., chapter 8.2 of [81]). In Petri nets they correspond to transition firing sequences (see, e.g., [66]).

The Petri net produces a language modulo the alphabet Σ that contains all the external events of $\mathcal{S}$. CSP's traces only contains events that are external (i.e., observable from outside the system). Therefore, this notion of equivalence implies that, if we ignore internal events such as τ, then the sequences of (observable) actions of both systems are exactly the same.

5.2 Transformation of a CSP Specification into an Equivalent Petri Net

This section introduces an algorithm to transform a CSP specification into an equivalent Petri net. In summary, the steps performed by the transformation are the following: firstly, the algorithm takes a CSP specification and executes the extended semantics with an empty store. The execution of the semantics produces a Petri net that represents the performed computation. When the computation is finished, the extended semantics returns to the algorithm a new store with the information about the choices that have been executed. Then, the algorithm determines with this information whether new computations not explored yet exist. If this is the case, the semantics is executed again with an updated store. This is repeated until all possible computations have been explored. This sequence of steps gradually augments the Petri net produced. When the algorithm detects that no more computations are possible (i.e., the store is empty), it outputs the current Petri net as the final result.

Even though the transformation is controlled by an algorithm, the generation of the final Petri net is carried out by an instrumented operational semantics of CSP. In particular, the algorithm fires the execution of the semantics that generates incrementally and as a side effect the Petri net.

The instrumentation of the semantics performs three main tasks:

1. It produces a computation and generates as a side-effect a Petri net associated with the computation.
2. It controls that no infinite loops are executed.
3. It ensures that the execution is deterministic.

Algorithm 5.1 drives the transformation, controls the execution of the semantics and repeatedly uses it to deterministically execute all possible computations—of the original (non-deterministic) specification—and the Petri net is constructed incrementally with each execution of the semantics. Namely, each time the semantics is executed, it produces as a result a portion of the Petri net. This result is the input of the next execution of the semantics that adds a new part of the Petri net. This process is repeated until all

Algorithm 5.1 General Algorithm

Input: A CSP specification $\mathcal{S}$ with initial process `MAIN`
Output: A labeled Petri net $\mathcal{N}$ equivalent to $\mathcal{S}$

Build the initial state of the semantics: $state = (\texttt{MAIN}, p_0, \mathcal{N}_0, ([], []), \varnothing)$
where $\mathcal{N}_0 = (\langle \{p_0\}, \varnothing, \varnothing \rangle, M_0, \mathcal{P}, \mathcal{T}, \mathcal{L}_P, \mathcal{L}_T)$, $M_0(p_0) = 1$,
$\mathcal{P} = Names \cup \{\square, \sqcap\}$, $\mathcal{T} = \Sigma^\tau \cup \{\|, \texttt{C1}, \texttt{C2}\}$.
repeat
  repeat
    Run the rules of the instrumented semantics with the state *state*
  until no more rules can be applied
  Get the new state $state = (\_, \_, \mathcal{N}, ([], S), \_)$
  $state = (\texttt{MAIN}, p_0, \mathcal{N}, (\texttt{UpdStore}(S), []), \varnothing)$
until $\texttt{UpdStore}(S) = []$
return $\mathcal{N}$
where function `UpdStore` is defined as follows:

$$\texttt{UpdStore}(S) = \begin{cases} (rule, rules \backslash \{rule\}) : S' & \text{if } S = (\_, rules) : S' \wedge rule \in rules \\ \texttt{UpdStore}(S') & \text{if } S = (\_, \varnothing) : S' \\ [] & \text{if } S = [] \end{cases}$$

possible executions have been explored and thus the complete Petri net has been produced.

The key point of the algorithm is the use of a store that records the actions that can be performed by the semantics. In particular, the store is an ordered list of elements that allows us to add and extract elements from the beginning and the end of the list; and it contains tuples of the form $(rule, rules)$ where

- *rule* indicates the rule that must be selected by the semantics in the next execution step, when different possibilities exist. They are indicated with intuitive abreviatures of the semantics rules. Thanks to *rule* the semantics is deterministic because it knows at every step what rule must be applied.
- *rules* is a set containing the other possible rules that can be selected with the current control. Therefore, *rules* records at every step all the possible rules not applied so that the algorithm will execute the semantics again with these rules.

Algorithm 5.1 uses the store to prepare each execution of the semantics in-

dicating the rules that must be applied at each step. For this, function `UpdStore` is used. It avoids to repeat the same computation with the semantics. When the semantics finishes, the algorithm prepares a new execution of the semantics with an updated store. This is repeated until all possible computations are explored (i.e., until the store is empty).

Taking into account that the original semantics can be non-terminating (it can produce infinite computations), the instrumented semantics could be also non-terminating if a loop-checking mechanism is not incorporated to ensure termination. In order to ensure termination of all computations, the instrumentation of the semantics incorporates a mechanism to stop the computation when the same process is repeated in the same context (i.e., the same control appears twice in a (sub)derivation of the semantics).

In the instrumented semantics a *state* is a tuple $(P, p, \mathcal{N}, (S, S_0), \Delta)$, where:

- P is the process to be evaluated (the *control*),
- p is the last place added to the Petri net $\mathcal{N}$,
- (S, S_0) is a tuple with two stores (where the empty store is denoted by []) that contains the rules to apply and the rules applied so far, and
- Δ is a set of references used to insert synchronizations in $\mathcal{N}$.

The basic idea of the Petri net construction is to generate the Petri net associated with the current control and connect this net to the last place added to $\mathcal{N}$.

The rules of the instrumented semantics are presented in Figure 5.1. A brief explanation for the most relevant rules of the semantics follows:

(Process Call - Sequential) In the instrumented semantics, there are two versions of the standard rule for process call. The first version is used when a process call is made in a sequential process. The second version is used for process calls made inside parallelism operators. The sequential version basically decides whether process P must be unfolded or not. This is done to avoid infinite unfolding of the same process. Once a (sequential) process has been unfolded once, it is not unfolded again. If the process has been previously unfolded, then we are in a loop, and P is marked as a loop with

(Process Call - Sequential)

$$\frac{}{(M, p, \mathcal{N}, (S, S_0), \_) \xrightarrow{\tau} (P, p', \mathcal{N}', (S, S_0), \varnothing)}$$

$$(P, p', \mathcal{N}') \;=\; \texttt{LoopCheck}(M, p, \mathcal{N})$$

$$\texttt{LoopCheck}(M, p, \mathcal{N}) = \begin{cases} (\circlearrowleft(M), p, \mathcal{N}[t \mapsto q_M]) & \text{if } \mathcal{N}[q_M \mapsto \_, t \mapsto p] \\ (rhs(M), p'', \mathcal{N}[p_M \mapsto t_\tau \mapsto p'']) & \text{otherwise} \end{cases}$$

(Process Call - Parallel)

$$\frac{}{(M_\diamondsuit, p, \mathcal{N}, (S, S_0), \_) \xrightarrow{\tau} (rhs(M), p', \mathcal{N}[p_M \mapsto t_\tau \mapsto p'], (S, S_0), \varnothing)}$$

(Prefixing)

$$\frac{}{(a \to P, p, \mathcal{N}, (S, S_0), \_) \xrightarrow{a} (P, p', \mathcal{N}[p \mapsto t_a \mapsto p'], (S, S_0), \{(p, t_a, p')\})}$$

(Choice)

$$\frac{}{(P \boxminus Q, p, \mathcal{N}, (S, S_0), \_) \xrightarrow{\tau} (P', p', \mathcal{N}', (S', S_0'), \varnothing)} \qquad \boxminus \in \{\Box, \sqcup\}$$

$$(P', p', \mathcal{N}', (S', S_0')) \;=\; \texttt{SelectBranch}(P \boxminus Q, p, \mathcal{N}, (S, S_0))$$

$$\texttt{SelectBranch}(P \boxminus Q, p, \mathcal{N}, (S, S_0)) = \begin{cases} (P, p', \mathcal{N}[p_\boxminus \mapsto t_{C1} \mapsto p'], (S', (\mathsf{C1}, \{\mathsf{C2}\}) : S_0)) \\ \qquad\qquad \text{if } S = S' : (\mathsf{C1}, \{\mathsf{C2}\}) \\ (Q, p', \mathcal{N}[p_\boxminus \mapsto t_{C2} \mapsto p'], (S', (\mathsf{C2}, \varnothing) : S_0)) \\ \qquad\qquad \text{if } S = S' : (\mathsf{C2}, \varnothing) \\ (P, p', \mathcal{N}[p_\boxminus \mapsto t_{C1} \mapsto p'], ([], (\mathsf{C1}, \{\mathsf{C2}\}) : S_0)) \quad \text{otherwise} \end{cases}$$

Figure 5.1: An instrumented operational semantics that generates a Petri net

the special symbol $\circlearrowleft$. This label avoids to unfold the process again because no rule is applicable. In this case, to represent the loop in the Petri net, we add a new arc from the last added transition to the place labelled with the corresponding specification position. If P has not been previously unfolded, then its right-hand side becomes the new control. Moreover, the new Petri net contains a place that represents the process call and a transition that represents the occurrence of event τ.

(Process Call - Parallel) When a process call is made inside a parallelism operator, it is always unfolded. We do not worry about infinite unfolding because the rules for synchronized parallelism already control non-termination. In order to distinguish between process calls made sequentially or in parallel, we use a special symbol $\diamondsuit$. Therefore, for simplicity, we assume that all process calls inside parallelisms are labeled with $\diamondsuit$, and thus, the semantics can decide what rule should be used.

(Choice) The only sources of non-determinism are choice operators (differ-

ent branches can be selected for execution) and parallel operators (different order of branches can be selected for execution). Therefore, every time the semantics executes a choice or a parallelism, they are made deterministic thanks to the information in the store. In the case of choices, both internal and external can be treated with a single rule. In this rule, a function is used to produce the new control and the new tuple of stores, by selecting a branch with the information of the store. Given a choice operation $P \,\square\, Q$ (or $P \sqcap Q$), if the last element of the store indicates that the first branch of the choice (C1) must be selected, then P is the new control. If the second branch must be selected (C2), the new control is Q. In any other case the store is empty, and thus this is the first time that this choice is evaluated. Then, we select the first branch (P is the new control) and we add $(\mathsf{C1}, \{\mathsf{C2}\})$ to the store indicating that C1 has been chosen, and the remaining option is C2. Moreover, this rule creates a new transition for each branch that represents the τ event.

(Synchronized Parallelism 1 and 2) The store determines what rule to use when a parallelism operator is in the control. If we are not in a loop (this is known because the same control has not appeared before, and the last element in the store is SP1, then (Synchronized Parallelism 1) is used. If it is SP2, (Synchronized Parallelism 2) is used. In a synchronized parallelism composition, both parallel processes can be intertwiningly executed until a synchronized event is found. Therefore, places and transitions for both processes can be added interwoven to the Petri net. Hence, the semantics needs to know in every state the references to be used in both branches. This is done by labeling each parallelism operator with a tuple of the form (p_1, p_2, Υ) where p_1 and p_2 are respectively the last places added to the left and right branches of the parallelism; and Υ records the controls of the semantics in order to avoid repetition (i.e., it is used to avoid infinite loops). In particular, Υ is a set of triples of the form: $(P1 \| P2, p_1, p_2)$ where $P1 \| P2$ is the control of a previous state of the semantics, and p_1, p_2 are the nodes in the Petri net associated with $P1$ and $P2$. This tuple is initialized to $(\bot, \bot, \varnothing)$ for every parallelism that is introduced in the computation. Here, we use symbol $\bot$ to denote an undefined place. The new label of the parallelism contains a new Υ that has been updated with the current control only if it does not contain any $\circlearrowleft$. This is done with a simple syntactic checking.

These rules develop the branches of the parallelism until they are finished or until they must synchronize. The parallelism operator is represented in the

(Synchronized Parallelism 1)

$$\frac{(P1, p_1', \mathcal{N}', (S', (\mathsf{SP1}, rules) : S_0), \_) \xrightarrow{e} (P1', p_1'', \mathcal{N}'', (S'', S_0'), \Delta)}{(P1 \underset{X}{\parallel_{lab}} P2, p, \mathcal{N}, (S' : (\mathsf{SP1}, rules), S_0), \_) \xrightarrow{e} (P1' \underset{X}{\parallel_{lab'}} P2, p, \mathcal{N}'', (S'', S_0'), \Delta)} \quad e \in \Sigma^\tau \backslash X$$

$lab = (p_1, p_2, \Upsilon) \ \wedge \ (P1 \| P2, \_, \_) \notin \Upsilon \ \wedge$

$(\mathcal{N}', p_1', p_2') \ = \ \texttt{InitBranches}(\mathcal{N}, p_1, p_2, p) \ \wedge \ lab' = (p_1'', p_2', \texttt{NewUpsilon}(\Upsilon, (P1 \| P2, p_1', p_2')))$

(Synchronized Parallelism 2)

$$\frac{(P2, p_2', \mathcal{N}', (S', (\mathsf{SP2}, rules) : S_0), \_) \xrightarrow{e} (P2', p_2'', \mathcal{N}'', (S'', S_0'), \Delta)}{(P1 \underset{X}{\parallel_{lab}} P2, p, \mathcal{N}, (S' : (\mathsf{SP2}, rules), S_0), \_) \xrightarrow{e} (P1 \underset{X}{\parallel_{lab'}} P2', p, \mathcal{N}'', (S'', S_0'), \Delta)} \quad e \in \Sigma^\tau \backslash X$$

$lab = (p_1, p_2, \Upsilon) \ \wedge \ (P1 \| P2, \_, \_) \notin \Upsilon \ \wedge$

$(\mathcal{N}', p_1', p_2') \ = \ \texttt{InitBranches}(\mathcal{N}, p_1, p_2, p) \ \wedge \ lab' = (p_1', p_2'', \texttt{NewUpsilon}(\Upsilon, (P1 \| P2, p_1', p_2')))$

(Synchronized Parallelism 3)

$$\frac{Left \quad Right}{(P1 \underset{X}{\parallel_{lab}} P2, p, \mathcal{N}, (S' : (\mathsf{SP3}, rules), S_0), \_) \xrightarrow{e} (P1' \underset{X}{\parallel_{lab'}} P2', p, \mathcal{N}_s, (S''', S_0''), \Delta)} \quad e \in X$$

$lab = (p_1, p_2, \Upsilon) \ \wedge \ (P1 \| P2, \_, \_) \notin \Upsilon \ \wedge \ (\mathcal{N}', p_1', p_2') \ = \ \texttt{InitBranches}(\mathcal{N}, p_1, p_2, p) \ \wedge$

$Left = (P1, p_1', \mathcal{N}', (S', (\mathsf{SP3}, rules) : S_0), \_) \xrightarrow{e} (P1', p_1'', \mathcal{N}'', (S'', S_0'), \Delta_1) \ \wedge$

$Right = (P2, p_2', \mathcal{N}'', (S'', S_0'), \_) \xrightarrow{e} (P2', p_2'', \mathcal{N}''', (S''', S_0''), \Delta_2) \ \wedge$

$lab' = (p_1'', p_2'', \texttt{NewUpsilon}(\Upsilon, (P1 \| P2, p_1', p_2'))) \ \wedge$

$\mathcal{N}_s = (\mathcal{N}''' \cup \{(p \mapsto t_e \mapsto p') \mid (p, \_, p') \in (\Delta_1 \cup \Delta_2)\}) \backslash \{(p \mapsto t \mapsto p') \mid (p, t, p') \in (\Delta_1 \cup \Delta_2)\}$

$\wedge \ \Delta = \{(p, t_e, p') \mid (p, \_, p') \in (\Delta_1 \cup \Delta_2)\}$

$$\texttt{InitBranches}(\mathcal{N}, p_1, p_2, p) = \begin{cases} (\mathcal{N}[p \mapsto t_\| \mapsto p_1', p \mapsto t_\| \mapsto p_2'], p_1', p_2') & \text{if } p_1 = \bot \\ (\mathcal{N}, p_1, p_2) & \text{otherwise} \end{cases}$$

$$\texttt{NewUpsilon}(\Upsilon, (P1 \| P2, p_1, p_2)) = \begin{cases} \Upsilon & \text{if } \texttt{HasLoops}(P1 \| P2) \\ \Upsilon \cup \{(P1 \| P2, p_1, p_2)\} & \text{otherwise} \end{cases}$$

Figure 5.1: An instrumented operational semantics that generates a Petri net (cont.)

Petri net with a transition $t_\|$. This transition is connected to two new places, one for each branch.

(Synchronized Parallelism 3) It is applied when the last element in the store is SP3 and no loop is detected. It is used to synchronize the parallel processes. In this rule, all the events that have been executed in this step must be synchronized. Therefore, all the events occurred in the subderivations of the paralyzed processes are mutually synchronized. This is done in the Petri net by removing the transitions that were added in each subderivation and connecting all of them with a single transition. The new synchronization set

contains all the synchronizations occurred in both branches connected by the new transition.

(Synchronized Parallelism 4)

$$\frac{}{(P1 \underset{X}{\parallel}_{(p_1,p_2,\Upsilon)} P2, p, \mathcal{N}, (S' : (\mathsf{SP4}, rules), S_0), \_) \xrightarrow{\tau} (P', p, \mathcal{N}', (S', (\mathsf{SP4}, rules) : S_0), \varnothing)}$$

$$(P', \mathcal{N}') = \texttt{LoopControl}(P1 \underset{X}{\parallel}_{(p_1,p_2,\Upsilon)} P2, \mathcal{N})$$

$$\texttt{LoopControl}(P1 \underset{X}{\parallel}_{(p_1,p_2,\Upsilon)} P2, \mathcal{N}) =$$

$$\begin{cases} ((\circlearrowleft(P1'' \underset{X}{\parallel}_{(p_1',p_2',\Upsilon)} P2''), \mathcal{N}) & \text{if } P1' = \circlearrowleft(P1'') \wedge (P2' = \circlearrowleft(P2'') \vee \\ & \quad ((P2' = \texttt{STOP} \vee (P1'' \| P2', \_, \_) \in \Upsilon) \ \wedge \ P2'' = P2')) \\ (P1''' \underset{X}{\parallel}_{(p_1',p_2',\Upsilon)} P2', \mathcal{N}') & \text{if } P1' = \circlearrowleft(P1'') \wedge P2' \neq \circlearrowleft(\_) \wedge P2' \neq \texttt{STOP} \wedge \\ & \quad (P1'' \| P2', \_, \_) \notin \Upsilon \ \wedge \ (P1''', \mathcal{N}') = \texttt{DelEdges}(P1'', \mathcal{N}) \\ (\texttt{STOP}, \mathcal{N}) & \text{otherwise} \end{cases}$$

where $(P1', p_1', P2', p_2') \in \{(P1, p_1, P2, p_2), (P2, p_2, P1, p_1)\}$

$$\texttt{DelEdges}(P1 \underset{X}{\parallel}_{(p_1,p_2,\Upsilon)} P2, \mathcal{N}) =$$

$$\begin{cases} (P1' \underset{X}{\parallel}_{(p_1,p_2,\Upsilon')} P2', \mathcal{N}'') & \text{if } (P1 \| P2, pp_1, pp_2) \in \Upsilon \wedge \Upsilon' = \Upsilon \backslash \{(P1 \| P2, pp_1, pp_2)\} \wedge \\ & \quad ((P1 \neq (\_\|\_) \wedge \mathcal{N}' = \mathcal{N}[t_1 \mapsto p_1] \backslash \{t_1 \mapsto pp1\} \wedge P1' = P1) \\ & \quad \vee (P1 = (\_\|\_) \wedge (P1', \mathcal{N}') = \texttt{DelEdges}(P1, \mathcal{N}))) \wedge \\ & \quad ((P2 \neq (\_\|\_) \wedge \mathcal{N}'' = \mathcal{N}'[t_2 \mapsto p_2] \backslash \{t_2 \mapsto pp2\} \wedge P2' = P2) \\ & \quad \vee (P1 = (\_\|\_) \wedge (P2', \mathcal{N}'') = \texttt{DelEdges}(P2, \mathcal{N}'))) \\ (P1 \underset{X}{\parallel}_{(p_1,p_2,\Upsilon)} P2, \mathcal{N}) & \text{otherwise} \end{cases}$$

Figure 5.1: An instrumented operational semantics that generates a Petri net (cont.)

(Synchronized Parallelism 4) This rule is applied when the last element in the store is SP4. It is used when none of the parallel processes can proceed (because they already finished, deadlocked or were labeled with $\circlearrowleft$). When a parallelism is labeled as a loop with $\circlearrowleft$, it can be unlabeled to unfold it once[1] in order to allow the other processes to continue. This happens when the looped process is in parallel with other process and the later is waiting to synchronize with the former. In order to perform the synchronization, both processes must continue, thus the loop is unlabeled. This task is done by function `LoopControl` that decides whether the branches of the parallelism should be further unfolded or they should be stopped (e.g., due to a deadlock or an infinite loop). This function can detect three different situations:

[1] Only once because it will be labeled again when the loop is repeated.

(Synchronized Parallelism 5)

$$\frac{(P,p,\mathcal{N}_P,(S'_P,S_0),\_) \xrightarrow{e} (P',p,\mathcal{N}',(S',S'_0),\Delta)}{(P1 \underset{X}{\parallel}_{(p_1,p_2,\Upsilon)} P2,p,\mathcal{N},(S,S_0),\_) \xrightarrow{e} (P'',p,\mathcal{N}'',(S'',S''_0),\Delta')} \quad e \in \Sigma^\tau$$

$$S = [] \ \vee \ ((S = (\_ : (\mathsf{SP1},\_)) \ \vee \ S = (\_ : (\mathsf{SP2},\_)) \ \vee \ S = (\_ : (\mathsf{SP3},\_))) \\ \wedge \ (P1 \underset{X}{\parallel} P2,\_,\_) \in \Upsilon)$$

$$\wedge (P,\mathcal{N}_P,S_P) = \texttt{CheckLoops}(P1 \underset{X}{\parallel}_{(p1,p2,\Upsilon)} P2,\mathcal{N})$$

$$\wedge((S = [] \wedge S_P = [] \wedge e = \tau \wedge (P'',\mathcal{N}'',(S'',S''_0),\Delta') = (P,\mathcal{N}_P,([],S_0),\varnothing)) \\ \vee ((S = [] \wedge S_P \neq [] \wedge S'_P = S_P) \vee (S \neq [] \wedge S'_P = S) \\ \wedge (P'',\mathcal{N}'',(S'',S''_0),\Delta') = (P',\mathcal{N}',(S',S'_0),\Delta)))$$

$$\texttt{CheckLoops}(P1 \underset{X}{\parallel}_{(p1,p2,\Upsilon)} P2,\mathcal{N}) = \\ \begin{cases} (\circlearrowleft(P1 \underset{X}{\parallel}_{(p1,p2,\Upsilon)} P2),\mathcal{N}''',[]) & \text{if } (P1 \| P2,pp_1,pp_2) \in \Upsilon \\ & \quad \wedge \ ((\mathcal{N}'' = \mathcal{N}[t_1 \mapsto p_1, t_1 \mapsto pp_1] \ \wedge \ P1 \neq (\_\|\_)) \\ & \qquad \vee \ (\mathcal{N}'' = \mathcal{N} \wedge P1 = (\_\|\_))) \\ & \quad \wedge \ ((\mathcal{N}''' = \mathcal{N}''[t_2 \mapsto p_2, t_2 \mapsto pp_2] \ \wedge \ P2 \neq (\_\|\_)) \\ & \qquad \vee \ (\mathcal{N}''' = \mathcal{N}'' \wedge P2 = (\_\|\_))) \\ ((P1 \underset{X}{\parallel}_{(p1,p2,\Upsilon)} P2),\mathcal{N}_1 \cup \mathcal{N}_2,S') & \text{otherwise} \end{cases}$$

$$\text{where } (P1',\mathcal{N}_1,S_1) = \begin{cases} \texttt{CheckLoops}(P1,\mathcal{N}) & \text{if } P1 = \_\|\_ \\ (P1,\mathcal{N},[]) & \text{otherwise} \end{cases}$$

$$(P2',\mathcal{N}_2,S_2) = \begin{cases} \texttt{CheckLoops}(P2,\mathcal{N}) & \text{if } P2 = \_\|\_ \\ (P2,\mathcal{N},[]) & \text{otherwise} \end{cases}$$

$$Rules = \texttt{AppRules}(P1' \underset{X}{\parallel} P2')$$

$$S' = \begin{cases} S_2 : (\{\mathsf{SP2}\},\varnothing) & \text{if } P1 = \_\|\_ \wedge P2 \neq \_\|\_ \wedge S_1 = [] \wedge Rules = \{\mathsf{SP2}\} \\ S_1 : (\{\mathsf{SP1}\},\varnothing) & \text{if } P1 \neq \_\|\_ \wedge P2 = \_\|\_ \wedge S_2 = [] \wedge Rules = \{\mathsf{SP1}\} \\ S' : (r, Rules \backslash \{r\}) & \text{if } P1 \neq \_\|\_ \wedge P2 \neq \_\|\_ \wedge \mathsf{SP4} \notin Rules \\ & \quad \wedge \ r \in Rules \wedge (\ (S' = S_1 \wedge r = \mathsf{SP1}) \\ & \qquad \vee \ (S' = S_2 \wedge r = \mathsf{SP2}) \\ & \qquad \vee \ (S' = S_2 \cdot S_1 \wedge r = \mathsf{SP3}) \ ) \\ [(\{\mathsf{SP4}\},\varnothing)] & \text{otherwise} \end{cases}$$

$$\texttt{AppRules}(P1 \underset{X}{\parallel} P2) = \begin{cases} \{\mathsf{SP1}\} & \text{if } \tau \in \texttt{FstEvs}(P1) \\ \{\mathsf{SP2}\} & \text{if } \tau \notin \texttt{FstEvs}(P1) \wedge \tau \in \texttt{FstEvs}(P2) \\ R & \text{if } \tau \notin \texttt{FstEvs}(P1) \wedge \tau \notin \texttt{FstEvs}(P2) \ \wedge \ R \neq \varnothing \\ \{\mathsf{SP4}\} & \text{otherwise} \end{cases}$$

$$\text{where } \begin{cases} \mathsf{SP1} \in R & \text{if } \exists e \in \texttt{FstEvs}(P1) \wedge e \notin X \\ \mathsf{SP2} \in R & \text{if } \exists e \in \texttt{FstEvs}(P2) \wedge e \notin X \\ \mathsf{SP3} \in R & \text{if } \exists e \in \texttt{FstEvs}(P1) \wedge \exists e \in \texttt{FstEvs}(P2) \wedge e \in X \end{cases}$$

$$\texttt{FstEvs}(P) = \begin{cases} \{a\} & \text{if } \ P = a \rightarrow Q \\ \varnothing & \text{if } \ P = \circlearrowleft Q \ \vee \ P = \texttt{STOP} \\ \{\tau\} & \text{if } \ P = M \ \vee \ P = Q \Box R \ \vee \ P = (\texttt{STOP} \underset{X}{\parallel} \texttt{STOP}) \\ & \quad \vee \ P = (\circlearrowleft Q \underset{X}{\parallel} \circlearrowleft R) \ \vee \ P = (\circlearrowleft Q \underset{X}{\parallel} \texttt{STOP}) \ \vee \ P = (\texttt{STOP} \underset{X}{\parallel} \circlearrowleft R) \\ & \quad \vee \ (P = (\circlearrowleft Q \underset{X}{\parallel} R) \wedge \texttt{FstEvs}(R) \subseteq X) \vee (P = (Q \underset{X}{\parallel} \circlearrowleft R) \wedge \texttt{FstEvs}(Q) \subseteq X) \\ & \quad \vee \ (P = Q \underset{X}{\parallel} R \wedge \texttt{FstEvs}(Q) \subseteq X \wedge \texttt{FstEvs}(R) \subseteq X \wedge \bigcap_{M \in \{Q,R\}} \texttt{FstEvs}(M) = \varnothing) \\ E & \text{otherwise, with } P = Q \underset{X}{\parallel} R \wedge E = (\texttt{FstEvs}(Q) \cup \texttt{FstEvs}(R)) \backslash \\ & \qquad (X \cap (\texttt{FstEvs}(Q) \backslash \texttt{FstEvs}(R) \cup \texttt{FstEvs}(R) \backslash \texttt{FstEvs}(Q))) \end{cases}$$

Figure 5.1: An instrumented operational semantics that generates a Petri net (cont.)

(i) The parallelism is in a loop. In this case, the whole parallelism is marked as a loop. This situation happens when one of the branches is marked as looped (with $\circlearrowleft$), and the other branch is also looped, or it already terminated (i.e., it is STOP), or the control of both branches of the parallelism have been repeated (i.e., they are in Υ).

(ii) The parallelism is not in a loop, and it should proceed. This situation happens when one of the branches is marked as looped, and the other branch is trying to synchronize with the first one. In this case, the branch marked as a loop should continue to allow the synchronization. Therefore, the loop symbol $\circlearrowleft$ is removed and the loop arcs added to the Petri net are also recursively removed.

(iii) The parallelism must be stopped. This happens for instance because both branches terminated, therefore, the whole parallelism is replaced by STOP, thus, stopping further computations.

(Synchronized Parallelism 5) This rule is used to detect loops (when the control has been repeated and thus it appears in Υ, and SP1, SP2 or SP3 is the last element in the store), and also to determine what rule must be applied (when the store is empty).

In order to control non-termination, this rule checks whether the current control or other parallelisms inside it has been already repeated in the computation (this is done with the information in Υ). If this is the case, then we are in a loop, and the parallelisms are labeled with the symbol $\circlearrowleft$; thus it cannot continue unless this symbol is removed by other parallel process that requires the unfolding of this process (to synchronize). In case of loop, this function also adds the corresponding loop arcs to the Petri net. If a loop is not detected in the control of the parallelism, then the parallelism continues normally as in the standard semantics. If a loop is detected, then a new control labeled with $\circlearrowleft$ is returned directly without performing any subderivation. Another important task performed by this rule is the preparation of the store. This function builds the new store indicating what rules must be applied in the following derivations, also for its internal parallelisms. Essentially, it decides what rules are applicable depending on the events that could happen in the next step. Additionally, it implicitly imposes an order in the execution, and this order avoids the repetition of redundant derivations. For instance, if both branches of a parallelism can fire event τ in any order, then it will be fired first in the first branch (using rule SP1) and then in the second branch (using rule SP2). This avoids multiple unnecessary executions such as SP1, SP2 and SP2, SP1 where only τ happens in both branches but in

different order. Therefore, rule (Synchronized Parallelism 5) prepares the store allowing the semantics to proceed with the correct rule.

Consider the Moore machine [41] in Figure 5.2 to compute the remainder of a binary number divided by three. The different values for the possible remainders are 0, 1 and 2. Note that if a decimal value n written in binary is followed by a 0 then its decimal value becomes $2n$ and if n is followed by a 1 then its value becomes $2n + 1$. If the remainder of $n/3$ is r, then the remainder of $2n/3$ is $2r \ mod \ 3$. If $r = 0$, 1, or 2, then $2r \ mod \ 3$ is 0, 2, or 1, respectively. Similarly, the remainder of $(2n + 1)/3$ is 1, 0, or 2, respectively. So, this machine has 3 states: q_0 is the start state and represents a remainder 0, state q_1 represents a remainder 1 and state q_2 represents a remainder 2.

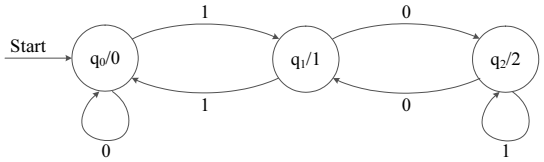


Figure 5.2: Moore machine to determine the remainder of a binary number divided by three

The CSP specification of Figure 5.3 corresponds to the previous Moore machine with a slight modification: the fact that a binary number is divisible by 3 is explicitly represented. Processes `REM0`, `REM1` and `REM2` are considered as remainder 0, 1 and 2 states, respectively. We know that a number n is divisible by 3 if the remainder of $n/3$ is 0. So, process `REM0` also represents that the number is divisible by 3 (represented with the event `divisible3`). Using this specification as input for the transformation algorithm, we obtain the Petri net drawn in Figure 5.4.

The CSP specification of Figure 5.5 is an extension of the previous CSP specification to check whether a given binary number is divisible by 3.

$$
\begin{aligned}
&\texttt{MAIN} = \texttt{REM0} \\
&\texttt{REM0} = (\texttt{0} \rightarrow \texttt{REM0}) \ \Box \ (\texttt{1} \rightarrow \texttt{REM1}) \ \Box \ (\texttt{divisible3} \rightarrow \texttt{STOP}) \\
&\texttt{REM1} = (\texttt{0} \rightarrow \texttt{REM2}) \ \Box \ (\texttt{1} \rightarrow \texttt{REM0}) \\
&\texttt{REM2} = (\texttt{0} \rightarrow \texttt{REM1}) \ \Box \ (\texttt{1} \rightarrow \texttt{REM2})
\end{aligned}
$$

Figure 5.3: CSP specification of the Moore Machine of Figure 5.2

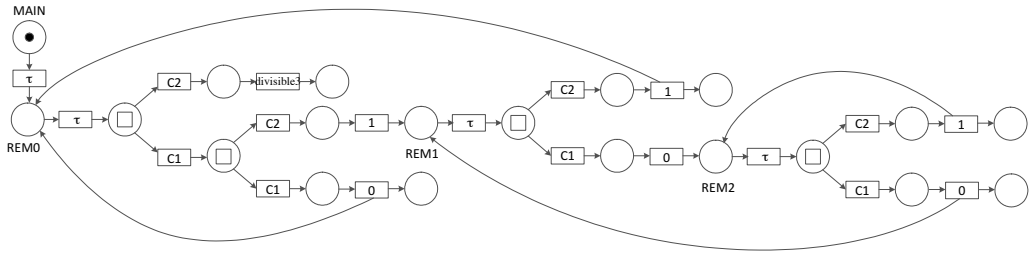


Figure 5.4: Petri net associated with the specification of Figure 5.3

$$\texttt{MAIN} = \texttt{REM0} \underset{\{0,1,\texttt{divisible3}\}}{\parallel} \texttt{BINARY}$$
$$\texttt{REM0} = (0 \rightarrow \texttt{REM0}) \,\Box\, (1 \rightarrow \texttt{REM1}) \,\Box\, (\texttt{divisible3} \rightarrow \texttt{STOP})$$
$$\texttt{REM1} = (0 \rightarrow \texttt{REM2}) \,\Box\, (1 \rightarrow \texttt{REM0})$$
$$\texttt{REM2} = (0 \rightarrow \texttt{REM1}) \,\Box\, (1 \rightarrow \texttt{REM2})$$
$$\texttt{BINARY} = 1 \rightarrow 1 \rightarrow 0 \rightarrow \texttt{divisible3} \rightarrow \texttt{STOP}.$$

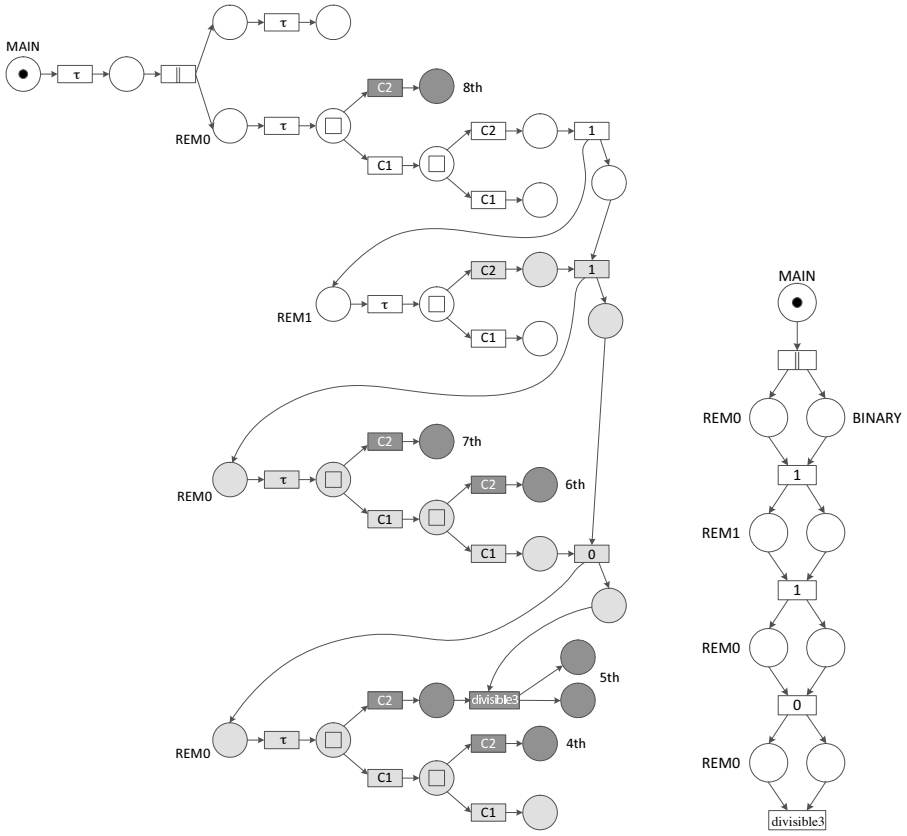
Figure 5.5: Extension of the specification of Figure 5.3

Process BINARY represents a binary number; in this case, the binary number 110 (which corresponds to the decimal value 6). Processes REM0 and BINARY are executed in parallel with $\{0, 1, \texttt{divisible3}\}$ as the set of synchronized events, i.e., whenever one of these synchronized events happens in process REM0, it must also happen in process BINARY at the same time, and vice versa. So, if event divisible3 occurs, it means that the binary number is divisible by 3. When the binary number is not divisible by 3, the remainder of its division between 3 will be 1 or 2 (processes REM1 or REM2), and then event divisible3 will never happen.

Due to the set of synchronized events $\{0, 1, \texttt{divisible3}\}$ in process MAIN and to the choice operators in processes REM0 and REM1, this specification can produce the set of finite sequences of observable events defined as:

$$\mathit{traces}(\texttt{MAIN}) = \{\langle\rangle, \langle 1\rangle, \langle 1, 1\rangle, \langle 1, 1, 0\rangle, \langle 1, 1, 0, \texttt{divisible3}\rangle\}$$

In particular, it can produce the sequence of events $\langle 1, 1, 0, \texttt{divisible3}\rangle$ before it is deadlocked. The transformation algorithm with this specificification produces the Petri net shown in Figure 5.6(a). This Petri net is generated after eight iterations of the algorithm (and thus eight executions of the instrumented semantics). We first execute the semantics with the initial state and get the computation that corresponds to the execution of the left branch



(a) PN generated with the transformation algorithm (b) PN optimized

Figure 5.6: PN associated with the specification of Figure 5.5

of the two choices of process `REM0`. The final state's store contains two pairs $(\texttt{C1}, \{\texttt{C2}\})$ to denote that the left branch of the choices has been executed and the right branch is still pending. Then, the algorithm calls the function that updates the store and executes the semantics again with a new initial state. After this execution the Petri net shown in Figure 5.7 has been computed. The first iteration generates the white nodes of Figure 5.7 and grey nodes are generated in the second iteration. Figure 5.6(a) shows the final Petri net generated where white nodes were generated in the first and second iterations, grey nodes were generated in the third iteration; and black nodes were generated in the rest of iterations (from fourth to eighth). The language produced by the labeled Petri net in Figure 5.6(a) is shown in Figure 5.8

If we restrict the language to visible events, we get the language over the alphabet Σ:

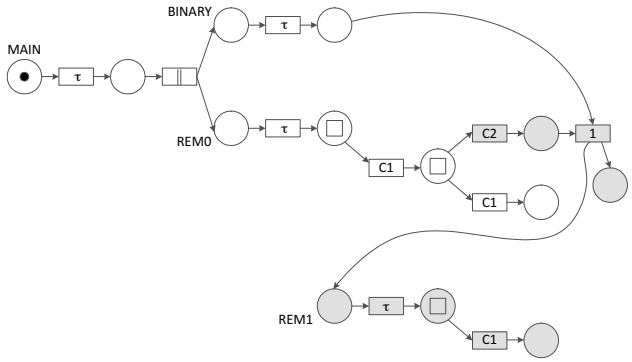


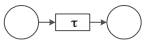
Figure 5.7: Petri net generated in the first and second iterations of the transformation algorithm for the specification of Figure 5.5

$$L_\Sigma(\mathcal{N}) = \{\langle\rangle, \langle \texttt{1}\rangle, \langle \texttt{1}, \texttt{1}\rangle, \langle \texttt{1}, \texttt{1}, \texttt{0}\rangle, \langle \texttt{1}, \texttt{1}, \texttt{0}, \texttt{divisible3}\rangle\}$$

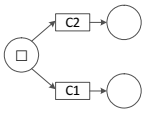
We can see that this language is exactly the same as the one produced by *traces*(`MAIN`). Then, according to the definition of equivalence, the Petri net generated by the transformation algorithm and the CSP specification of the example are equivalent.

With the previous examples, it is easy to see that the Petri nets generated by the transformation algorithm are very close to the CSP's semantics behavior. These Petri nets follow step by step the sequence of events (both internal and external) that happened during the evaluation of the semantics. For instance,

- Each occurrence of a τ is explicitly represented in the Petri net with a transition (labeled with the internal event τ) between two places:



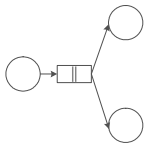
- Each choice is represented with a place labeled with the choice operator ($\square$ or $\sqcup$) and two transitions labeled with the first option (`C1`) and the second option (`C2`):



- Each parallelism operator is represented with a transition labeled with the parallelism operator || and two places:

$$
\begin{aligned}
L(\mathcal{N}) = \{&\langle\rangle, \langle\tau\rangle, \langle\tau,\ ||\rangle, \langle\tau,\ ||,\ \tau\rangle, \langle\tau,\ ||,\ \tau,\ \tau\rangle, \langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C2}\rangle, \langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1}\rangle,\\
&\langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2}\rangle, \langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C1}\rangle, \langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ 1\rangle,\\
&\langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ 1,\ \tau\rangle, \langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ 1,\ \tau,\ \texttt{C1}\rangle,\\
&\langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ 1,\ \tau,\ \texttt{C2}\rangle, \langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ 1,\ \tau,\ \texttt{C2},\ 1\rangle,\\
&\langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ 1,\ \tau,\ \texttt{C2},\ 1,\ \tau\rangle,\\
&\langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ 1,\ \tau,\ \texttt{C2},\ 1,\ \tau,\ \texttt{C2}\rangle,\\
&\langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ 1,\ \tau,\ \texttt{C2},\ 1,\ \tau,\ \texttt{C1}\rangle,\\
&\langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ 1,\ \tau,\ \texttt{C2},\ 1,\ \tau,\ \texttt{C1},\ \texttt{C2}\rangle,\\
&\langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ 1,\ \tau,\ \texttt{C2},\ 1,\ \tau,\ \texttt{C1},\ \texttt{C1}\rangle,\\
&\langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ 1,\ \tau,\ \texttt{C2},\ 1,\ \tau,\ \texttt{C1},\ \texttt{C1},\ 0\rangle,\\
&\langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ 1,\ \tau,\ \texttt{C2},\ 1,\ \tau,\ \texttt{C1},\ \texttt{C1},\ 0,\ \tau\rangle,\\
&\langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ 1,\ \tau,\ \texttt{C2},\ 1,\ \tau,\ \texttt{C1},\ \texttt{C1},\ 0,\ \tau,\ \texttt{C1}\rangle,\\
&\langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ 1,\ \tau,\ \texttt{C2},\ 1,\ \tau,\ \texttt{C1},\ \texttt{C1},\ 0,\ \tau,\ \texttt{C1},\ \texttt{C1}\rangle,\\
&\langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ 1,\ \tau,\ \texttt{C2},\ 1,\ \tau,\ \texttt{C1},\ \texttt{C1},\ 0,\ \tau,\ \texttt{C1},\ \texttt{C2}\rangle,\\
&\langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ 1,\ \tau,\ \texttt{C2},\ 1,\ \tau,\ \texttt{C1},\ \texttt{C1},\ 0,\ \tau,\ \texttt{C2}\rangle,\\
&\langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ 1,\ \tau,\ \texttt{C2},\ 1,\ \tau,\ \texttt{C1},\ \texttt{C1},\ 0,\ \tau,\ \texttt{C2},\ \texttt{divisible3}\rangle\}
\end{aligned}
$$

Figure 5.8: Language produced by the PN in Fig. 5.6(a)



These properties make the generated Petri net to be compositional, and it is easy to see that the Petri net is a graphical representation of the CSP's semantics derivations. In fact, this is a powerful tool for program comprehension because the Petri net is very similar to the so-called *Control Flow Graph* (CFG) (see, e.g., [86]) of imperative languages. Note that the paths followed by tokens are the possible execution paths in the semantics.

5.3 Tuning the Generated Petri Net

For some applications, we may be interested in producing a Petri net as small as possible discarding internal events and only concentrating on external ones. This simplified Petri net should keep the equivalence with the CSP specification but still reduce its size. However, (part of) the connection with

the CSP semantics behavior would be lost. This section introduces a transformation for Petri nets that can be applied to the Petri nets generated by our technique. The transformation takes advantage of the particular shape of the generated Petri nets to remove unnecessary parts that do not contribute to the language produced by them.

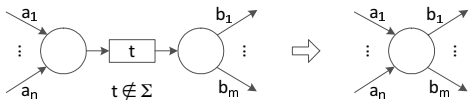
Because both versions of the Petri net (the complete and the simplified) are useful, we decided not to generate the simplified version directly from the instrumented semantics, and do it with a post-process transformation. This has the additional advantage of not introducing more complexity in the instrumentation of the semantics. As an example consider the complete Petri net in Figure 5.6(a) and its simplified version in Figure 5.6(b).

The algorithm that performs the simplification process of Petri nets is introduced in Paper 4. This algorithm takes a Petri net and iteratively deletes all parts of the net that are duplicated or useless, until a fix-point is reached (i.e., no more simplifications can be done).

In order to delete duplicates, the algorithm traverses the Petri net from the initial place (labeled with `MAIN`) following all paths. During the traversal, it tries to identify repeated parts of the Petri net to remove the repetitions and reuse one of them, whenever it is possible.

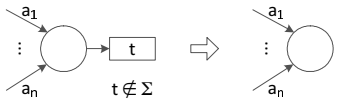
The task of deleting useless parts of the Petri net such as sequences of linear non-observable transitions is independent of the previous algorithm. This task is performed by a function that removes useless nodes by checking those transitions that do not contribute to the final trace. We call these transitions *Candidates* and they are initially those transitions labeled with τ, `C1` and `C2`. The function checks whether a sequence of transitions of this kind exists, and if so, they are removed. For instance, some clear opportunities for optimization are the following:

- Removing useless transitions:



- Removing sink places:

- Removing final non-observable transitions:



In Figure 5.9 we show the optimized Petri net associated with the specification of Figure 5.3. This Petri net is the output produced by the tuning algorithm with the Petri net in Figure 5.4 as input.

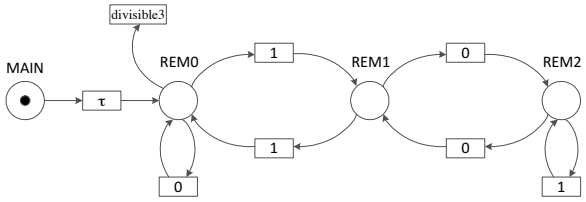


Figure 5.9: PN optimized associated with the specification of Figure 5.3

Turning back to the specification in Figure 5.5, its associated Petri net in Figure 5.6(a) is optimized by the tuning algorithm producing the Petri net in Figure 5.6(b). Observe that in this simplified Petri net it has been made clearly explicit the parallel execution of process `BINARY` with the sequential execution of processes `REM0` and `REM1`. It is also clear that event `divisible3` can only happen if processes `REM0`, `REM1`, `REM0` and `REM0` are executed sequentially and they synchronize on events `1`, `1`, `0` and `divisible3` with the parallel execution of process `BINARY`.

5.4 Implementation

All the algorithms proposed and the instrumented operational semantics have been implemented and integrated into a tool called *CSP2PN*. This tool allows us to automatically generate a Petri net equivalent to a given CSP specification. The tool has been implemented in Prolog and C. It has about 1800 LOC and generates Petri nets in the standard `PNML` format [78] (it can also generate Petri nets in `dot` and `jpg` formats). Although *CSP2PN* implements the technique described in this paper, the implemented algorithm is much more complex due to efficiency reasons.

In particular, the implementation of the algorithm contains some improvements that significantly speed up the Petri net construction. The most important improvement is to avoid repeated computations. This is done by: (i) state memoization: once a state already explored is reached the algorithm stops this computation and starts with another one; and (ii) skipping already performed computations: computations do not start from `MAIN`, they start from the next non-deterministic state in the execution (this is provided by the information of the store).

The implementation is composed of eight different modules that interact to produce the final Petri net:

Main: This is the main module that coordinates all the other modules.

Control Algorithm: Implements the transformation algorithm and the data structures needed to communicate with the semantics (e.g., the store).

Semantics: Implements the CSP's extended operational Semantics.

Optimization: Implements all the optimization technique (the tuning algorithm).

Pretty Printing: All the derivations performed by the semantics and the logs of execution are printed with this module.

Graph Generation: It produces the final Petri nets with different formats such as DOT and JPG.

PNML Construction: This is the only module written in C (the others are written in Prolog). It basically reads the generated Petri net in DOT format and transforms it into a standard PNML Petri net.

Tools: It contains common and auxiliary functions and tools used by the other modules.

The implementation, source code and several examples are publicly available at [57]. There is an online version of *CSP2PN* that can be used to test the tool. This online version is publicly available at [58].

Figure 5.10 shows a screenshot of the online version of *CSP2PN*. You can either write down the initial CSP specification or choose one from the list of available examples. Once the Petri net is generated (Generate Petri net)

it is possible to visualize it (View Petri net) and to save it as `pnml`, `jpg` or `dot` formats. The same options are available for the optimized Petri net. For instance, the Petri net in Figure 5.4 has been automatically generated by *CSP2PN* from the CSP specification of Example 5.3. After the Petri net is generated, we also show the execution log of the instrumented semantics used by the transformation technique. This log allows us to check the different iterations of the algorithm and to follow the execution of the instrumented semantics step by step.

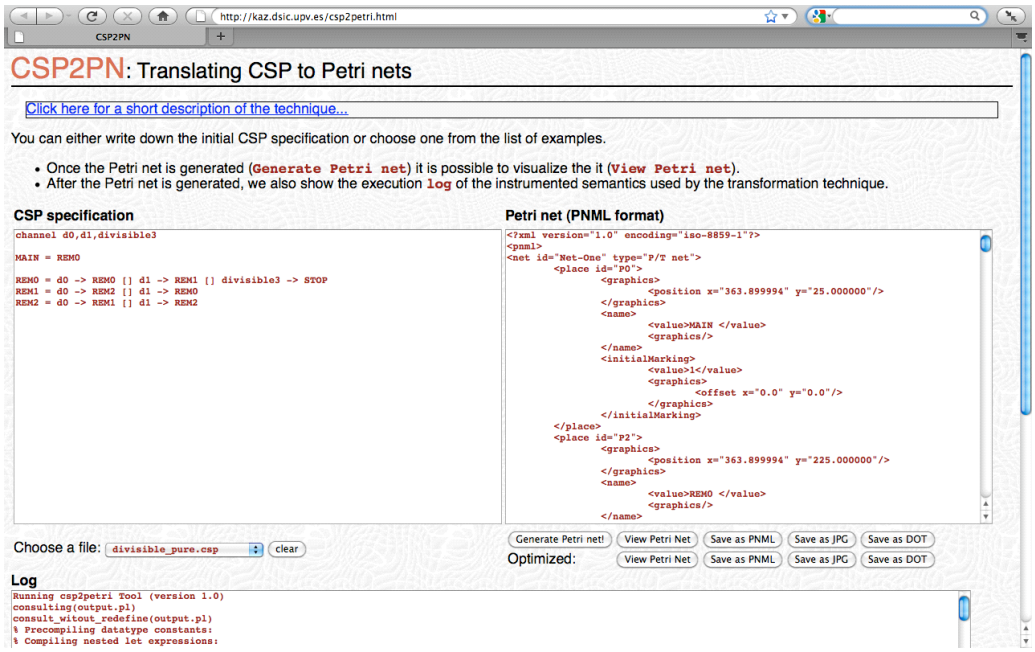


Figure 5.10: Screenshot of the online version of *CSP2PN*

The possibility of saving the generated Petri net as a `pnml` file allows us to animate and analyze it with any standard Petri net tool. The results of these analyses can be transferred easily to the CSP specification. For instance, *PIPE2* (Platform Independent Petri net Editor 2) [69] is a tool for creating and analysing Petri nets that loads and saves nets in `pnml`. Therefore, the optimized Petri nets generated by *CSP2PN* can be directly verified with the analyses performed by *PIPE2*. Figure 5.11 shows a screenshot of PIPE2 with a Petri net generated by CSP2PN and one of the possible verification analyses performed by PIPE2.

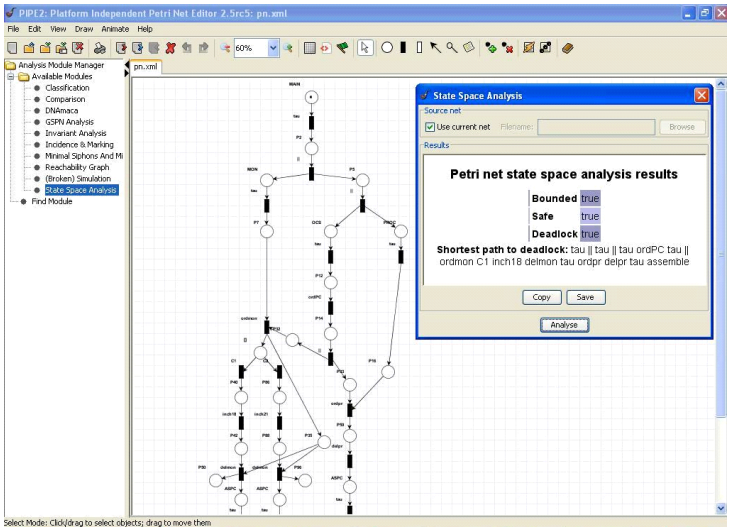


Figure 5.11: Screenshot of PIPE2

5.5 Related Work

We can group all previous approaches aimed at transforming CSP to Petri nets into two major research lines. The first line is based on traces describing the behavior of the system. In [62], starting from a trace-based representation of the behavior of the system, according to a subset of the Hoare's theory where no sequential composition with recursion is allowed, a stochastic Petri net model is built in a modular and systematic way. The overall model is built by modeling the system's components individually, and then putting them together by means of superposition. The second line of research includes all methodologies that translate CSP specifications into Petri nets directly from the CSP syntax. One of the first works translating CSP to Petri nets was [27], where distributed termination is assumed but nesting of parallel commands is not allowed. In [32], a CSP-like language is considered and translated into a subclass of Pr/T nets with individual tokens, where neither nesting of parallel commands is allowed nor distributed termination is taken into account. Other papers in this area are [71] that considers a subset of CCSP (the union of Milner's CCS[64] and Hoare's CSP[39]), and [28] which provides full CSP with a truly concurrent and distributed operational semantics based on Condition/Event Systems. There are also some works that translate process algebras into stochastic or timed Petri nets in order to perform real-time analyses and performance evaluation. Notable examples are [83, 61] that translate CSP specifications and [79] that define a composi-

tional stochastic Petri net semantics for the stochastic process algebra PEPA [38]. Even though this work is essentially different from ours because it is based on different formalisms, its implementation [10] is somehow similar to ours because the translation from PEPA to stochastic Petri nets is completely automatic. As in our work, all these papers do not allow recursion of nested parallel processes because the set of places of the generated Petri net would be infinite. In some way, our new semantics-based approach opens a third line of research where the transformation is directed by the semantics.

Chapter 6

Dynamic Slicing Techniques for Petri Nets

In this chapter, we propose the use of program slicing techniques to produce subnets of a Petri net. Program slicing has a great potential here since it allows us to syntactically reduce a model in such a way that the reduced model is composed only of those parts that may influence the slicing criterion. In the context of Petri nets, computing a net slice can be seen as a graph reachability problem. We propose two different alternatives for dynamic slicing of Petri nets that can be useful to reduce the size of the considered net, thereby simplifying subsequent analysis and debugging tasks by standard Petri net techniques. Firstly, we present a slicing technique that extends the slicing criterion in [74, 75] in order to also consider an initial marking. We show that this information can be very useful when analyzing Petri nets and, moreover, it allows us to significantly reduce the size of the computed slice. Furthermore, we show that our algorithm is, in the worst case, as precise as algorithms of previous aproaches. This can still be seen as a lightweight approach to slicing since its cost is bounded by the number of transitions in the Petri net. Then, we present a second approach that further reduces the size of the computed slice by only considering a particular execution—here, a sequence of transition firings. Clearly, in this case the computed slice is only useful to analyze the considered firing sequence.

6.1 Dynamic Slicing of Petri Nets

We say that our slicing technique is *dynamic* since an initial marking is taken into account (in contrast to previous approaches, e.g., [19, 48, 74, 75]). Using an initial marking can be useful, e.g., for debugging. Consider for instance that the user is analyzing a particular trace for a marked Petri net (using a simulation tool [37], which we assume correct), so that an erroneous state is reached. Here, by *erroneous* state, we mean a marking in which some places have an incorrect number of tokens. In this case, we are interested in extracting the set of places and transitions (more formally, a subnet) that may erroneously contribute tokens to the places of interest, so that the user can more easily locate the bug.

Thus, a slicing criterion is a tuple $\langle M_0, Q \rangle$ where M_0 is the initial marking and Q is the set of places of interest. Given a slicing criterion for a Petri net $\mathcal{N}$, we are interested in extracting a subnet with those places and transitions of $\mathcal{N}$ which can contribute to change the marking of Q in any execution starting in M_0. A Petri net $\mathcal{N}'$ is a slice of another Petri net $\mathcal{N}$ if $\mathcal{N}'$ is a subnet of $\mathcal{N}$ (i.e., no additional places nor transitions are added) and the behaviour of $\mathcal{N}$ is preserved in $\mathcal{N}'$ for the restricted sets of places and transitions. Trivially, given a Petri net $\mathcal{N}$, the complete net $\mathcal{N}$ is always a correct slice w.r.t. any slicing criterion. The challenge then is to produce a slice as small as possible.

Algorithm 6.1 describes our method to extract a dynamic slice from a Petri net. Intuitively speaking, Algorithm 6.1 constructs the slice of a Petri net (P, T, F) for a set of places $Q \subseteq P$ as follows. The key idea is to capture the token flow on places in Q. For this purpose,

- we first compute the possible paths that lead to the slicing criterion,
- then we also compute the paths that may be followed by the tokens of the initial marking.

This can be done by taking into account that

(i) the marking of a place p depends on its input and output transitions,

(ii) a transition may only be fired if it is enabled, and

(iii) the enabling of a transition depends on the marking of its input places.

Algorithm 6.1 Dynamic slicing of a marked Petri net.

Let $\mathcal{N} = (P, T, F)$ be a Petri net and let $\langle M_0, Q \rangle$ be a slicing criterion for $\mathcal{N}$. First, we compute a *backward slice* similar to that of [74]. This is obtained from $\mathsf{b\_slice}_{\mathcal{N}}(Q, \{\,\})$, where function $\mathsf{b\_slice}_{\mathcal{N}}$ is defined as follows:

$$\mathsf{b\_slice}_{\mathcal{N}}(W, W_{done}) = \begin{cases} \{\,\} & \text{if } W = \{\,\} \\ T \cup {}^\bullet T \cup \mathsf{b\_slice}_{\mathcal{N}}(W \setminus W'_{done}, W'_{done}) & \\ \qquad \text{if } W \neq \{\,\}, \text{ where } T = {}^\bullet p, \text{ and } W'_{done} = W_{done} \cup \{p\} & \\ \qquad\qquad \text{for some } p \in P & \end{cases}$$

Now, we compute a *forward slice* from

$$\mathsf{f\_slice}_{\mathcal{N}}(\{p \in P \mid M_0(p) > 0\}, \{\,\}, \{t \in T \mid M_0 \xrightarrow{t}\})$$

where function $\mathsf{f\_slice}_{\mathcal{N}}$ is defined as follows:

$$\mathsf{f\_slice}_{\mathcal{N}}(W, R, V) = \begin{cases} W \cup R & \text{if } V = \{\,\} \\ \mathsf{f\_slice}_{\mathcal{N}}(W \cup V^\bullet, R \cup V, V') & \\ \quad \text{if } V \neq \{\,\}, \text{ where } V' = \{t \in T \setminus (R \cup V) \mid {}^\bullet t \subseteq W \cup V^\bullet\} & \end{cases}$$

Then, the dynamic slice is finally obtained from the intersection of the backward and forward slices. Formally, let

$$P' \cup T' = \mathsf{b\_slice}_{\mathcal{N}}(Q, \{\,\}) \cap \mathsf{f\_slice}_{\mathcal{N}}(\{p \in P \mid M_0(p) > 0\}, \{\,\}, \{t \in T \mid M_0 \xrightarrow{t}\})$$

with $P' \subseteq P$ and $T' \subseteq T$, the computed slice is

$$\mathcal{N}' = (P', T', F|_{(P', T')})$$

The algorithm is divided into three steps:

- The first step is a backward slicing method (which is similar to the *basic slicing algorithm* of [74]) that obtains a slice $\mathcal{N}_1 = (P_1, T_1, F_1)$ defined as the subnet of $\mathcal{N}$ that includes all input places of all transitions connected to any place p in P_1, starting with $Q \subseteq P_1$.

- The second step is a forward slicing method that obtains a slice $\mathcal{N}_2 = (P_2, T_2, F_2)$ defined as the subnet of $\mathcal{N}$ that includes all transitions initially enabled in M_0 as well as those transitions connected as output transitions of places in P_2, starting with $p \in P$ such that $M_0(p) > 0$.

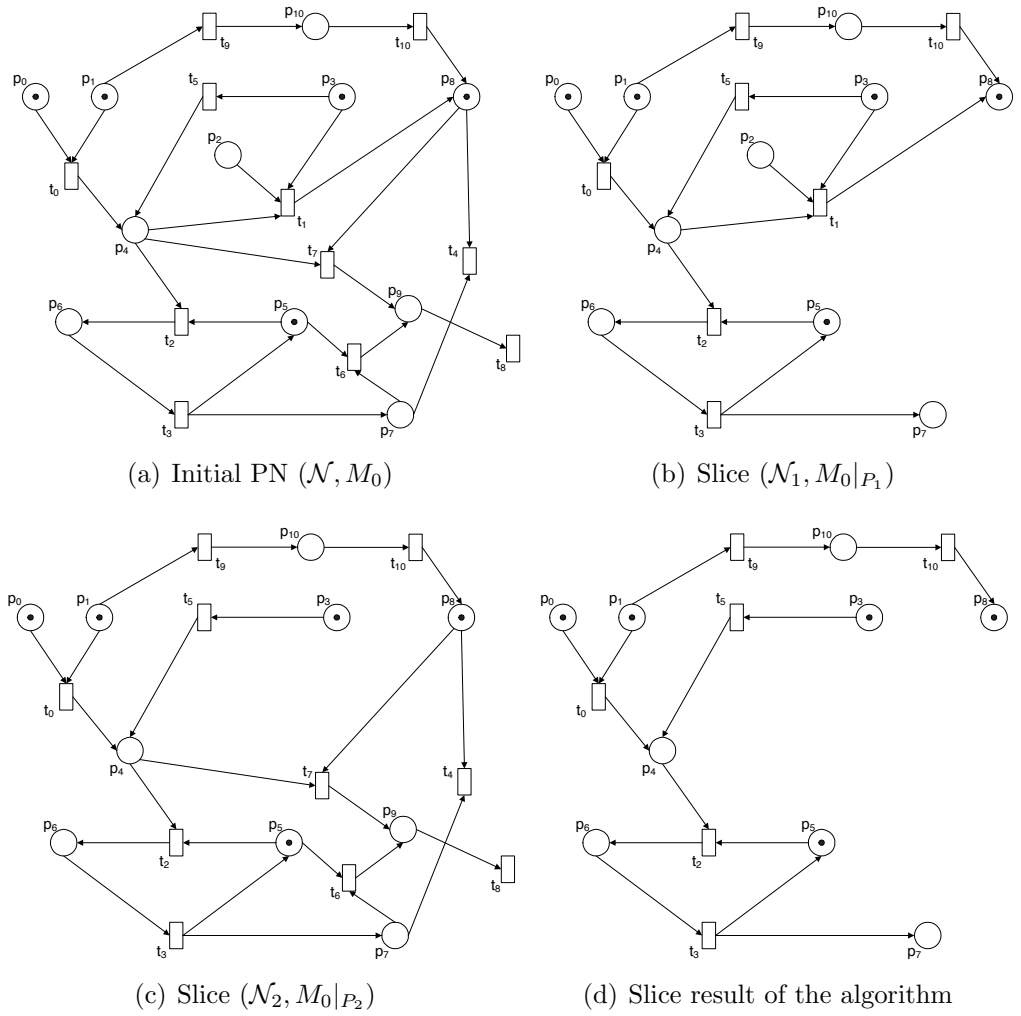
- Finally, the third step obtains the slice $\mathcal{N}' = (P', T', F')$ defined as the subnet of $\mathcal{N}$ where P' is the intersection of P_1 and P_2, T' is the intersection of T_1 and T_2, and F' is the restriction of F over P' and T', i.e., the intersection of backward and forward slices.

Consider for example the Petri net $\mathcal{N}$ of Fig. 6.1(a) where the user wants to produce a slice w.r.t. the slicing criterion $\langle M_0, \{p_5, p_7, p_8\}\rangle$. Figure 6.1(b) shows the slice $\mathcal{N}_1$ obtained in the first part of the algorithm. Figure 6.1(c) shows the slice $\mathcal{N}_2$ obtained in the second part of the algorithm. The subnet shown in Fig. 6.1(d) is the final result of the algorithm (the intersection of $\mathcal{N}_1$ and $\mathcal{N}_2$). This slice contains all the places and transitions of the original Petri net that can transmit tokens to the slicing criterion.

6.2 Extracting Slices from Traces

In this section, we present an alternative approach to dynamic slicing that generally produces smaller slices by also considering a particular firing sequence.

In principle, Algorithm 6.1 should consider all possible executions of the Petri net starting from the initial marking.This approach can be useful in several static contexts but it is too imprecise for debugging when a particular simulation has been performed. Therefore, in our second approach, we refine the notion of slicing criterion so as to also include the firing sequence that represents the erroneous simulation. By exploting this additional information, the new slicing algorithm will usually produce smaller slices.



(a) Initial PN $(\mathcal{N}, M_0)$

(b) Slice $(\mathcal{N}_1, M_0|_{P_1})$

(c) Slice $(\mathcal{N}_2, M_0|_{P_2})$

(d) Slice result of the algorithm

Figure 6.1: Example of an application of Algorithm 6.1

Now, a slicing criterion is a tuple $\langle M_0, \sigma, Q \rangle$ where σ represents the set of problematic firing sequences. Giving a slicing criterion for a Petri net, we are interested in extracting a subnet with those places and transitions which are necessary to move tokens to the places in Q. Trivially, given a marked Petri net $(\mathcal{N}, M_0)$, the complete net $\mathcal{N}$ is always a correct slice w.r.t. any slicing criterion. The challenge then is to produce a slice as small as possible.

Therefore, we have defined an algorithm to allow extracting slices from traces. Given a slicing criterion $\langle M_0, \sigma, Q \rangle$, the slicing algorithm proceeds as follows:

- The core of the algorithm lies in an auxiliary function that is initially

Algorithm 6.2 Extracting slices from traces.

Let $\mathcal{N} = (P, T, F)$ be a Petri net and let $\langle M_0, \sigma, Q \rangle$ be a slicing criterion for $\mathcal{N}$, with $\sigma = t_1 t_2 \dots t_n$. Then, we compute a dynamic slice $\mathcal{N}'$ of $\mathcal{N}$ w.r.t. $\langle M_0, \sigma, Q \rangle$ as follows:

- We have $\mathcal{N}' = (P', T', F')$, where $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \dots \xrightarrow{t_n} M_n$, $P' \cup T' = \mathsf{slice}(M_n, \sigma, Q)$, $P' \subseteq P$, $T' \subseteq T$, and $F' = F|_{(P',T')}$. Auxiliary function slice is defined as follows:

$$\mathsf{slice}(M_i, \sigma, W) = \begin{cases} W & \text{if } i = 0 \\ \mathsf{slice}(M_{i-1}, \sigma, W) & \text{if } \forall p \in W.\ M_{i-1}(p) \geqslant M_i(p),\ i > 0 \\ \{t_i\} \cup \mathsf{slice}(M_{i-1}, \sigma, W \cup {}^\bullet t_i) & \text{if } \exists p \in W.\ M_{i-1}(p) < M_i(p),\ i > 0 \end{cases}$$

- The initial marking M_0' is the restriction of M_0 over P', i.e., $M_0' = M_0|_{P'}$.

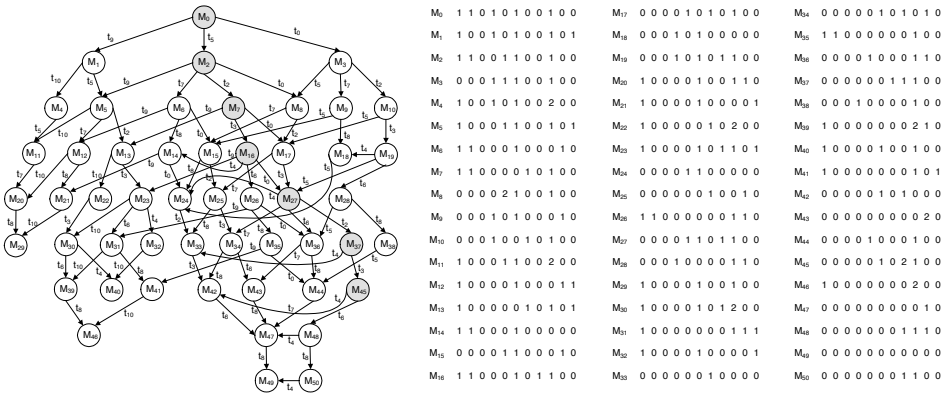
called with the marking M_n which is reachable from M_0 through σ, together with the firing sequence σ and the set of places Q of the slicing criterion.

- For a particular marking M_i, $i > 0$, a firing sequence σ and a set of places W, the function just moves "backwards" when no place in W increased its tokens by the considered firing.

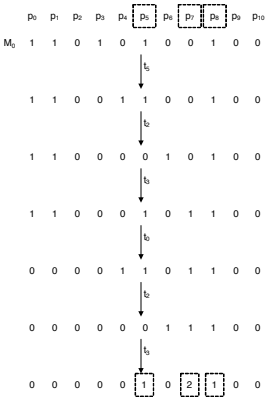
- Otherwise, the fired transition t_i increased the number of tokens of some place in W. In this case, the function already returns this transition t_i and, moreover, it moves backwards also adding the places in ${}^\bullet t_i$ to the previous set W.

- Finally, when the initial marking is reached, the function returns the accumulated set of places (which includes the initial places in Q).

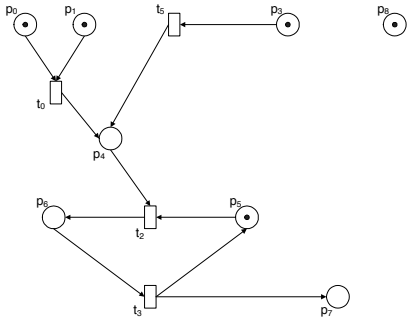
Consider for example the Petri net $\mathcal{N}$ shown in Fig. 6.1(a), together with the firing sequence σ shown in Fig. 6.2(b). The firing sequence $\sigma = t_5 t_2 t_3 t_0 t_2 t_3$ corresponds to the branch of the reachability graph shown in Fig. 6.2(a) that goes from the root to the node M_{45}. Then, the user can define the slicing criterion $\langle M_0, \sigma, \{p_5, p_7, p_8\} \rangle$ for $\mathcal{N}$; where M_0 is the initial marking for $\mathcal{N}$ defined in Fig 6.1(a). Clearly, this slicing criterion focus on a particular execution and thus the slice produced is more precise than the one produced by the algorithm for dynamic slicing of a marked Petri net. In this case, the slice of $\mathcal{N}$ w.r.t. $\langle M_0, \sigma, \{p_5, p_7, p_8\} \rangle$ is the Petri net shown in Fig. 6.2(c).



(a) Reachability graph



(b) Firing sequence σ



(c) Slice of $\mathcal{N}$ w.r.t. $\langle M_0, \sigma, \{p_5, p_7, p_8\}\rangle$

Figure 6.2: Example of an application of Algorithm 6.2

6.3 Related Work

Since it was originally defined by Weiser, program slicing has been applied to different formalisms that are not strictly programming languages, such as attribute grammars [85], hierarchical state machines [36], Z and CSP-OZ specifications [20, 12, 14], etc. Unfortunately, very little work has been carried out on slicing for Petri nets (some notable exceptions are [19, 48, 74, 75]). For instance, Chang and Wang [19] present a static slicing algorithm for Petri nets that slices out all sets of paths, known as concurrence sets, so that all paths within the same set should be executed concurrently. In [48], a static slicing technique for Petri nets is proposed in order to divide enormous Place/Transition nets (that are often regarded as low-level Petri

nets) into manageable modules so that the divided model can be analyzed by a compositional reachability analysis technique. A Petri net model is partitioned into concurrent units (Petri net slices) using minimal invariants. In order to preserve all the information in the original model, uncovered places should be added into minimally-connectable concurrent units since minimal invariants may not cover all the places. Finally, in [74, 75], Rakow presents another static slicing technique to reduce the Petri net size and, thus, lessen the problem of state explosion that occurs in the *model checking* [26] of Petri nets [6].

From the best of our knowledge, there is no previous proposal for *dynamic* slicing of Petri nets. This is surprising because, as we state previously, considering an initial marking and/or a particular sequence of transition firings allow us to further reduce the size of the slices and focus on a particular use of the considered Petri net. For instance, the slice of Fig. 6.1(b) is a subset of the slice produced by Rakow's algorithm [74] (this algorithm would also include transitions t_4, t_6 and t_7). Clearly, this slice contains parts of the Petri net that cannot be reached with the given initial marking (e.g., transition t_1 that could never be fired because place p_2 is empty). Rakow's algorithm computes all the parts of the Petri net that could transmit tokens to the slicing criterion and, thus, the associated slicing criterion is just $\langle Q \rangle$, where $Q \subseteq P$ is a set of places. In contrast, we compute all the parts of the Petri net that could transmit tokens to the slicing criterion from the initial marking. Therefore, our technique is essentially a generalization of Rakow's technique because the slice produced with Rakow's algorithm w.r.t. $\langle Q \rangle$ is the same as the slice produced w.r.t. $\langle M_0, Q \rangle$ if $M_0(p) > 0$ for all $p \in P$ and all $t \in T$ are enabled transitions at M_0. At the same time, it keeps its simplicity and efficiency because we still use the Petri net structure to produce the slice. As Rakow states in her PhD thesis [76], our algorithm is more aggressive (the sliced net is smaller) and it is not concerned with conserving safety properties. Note that the last is not necessary in several analysis. She also states that both variants produce the same results on strongly-connected nets. Therefore, our first approach can be considered *lightweight* because its cost is bounded by the number of transitions T of the original Petri net; namely, the cost of our algorithm is $\mathcal{O}(2T)$.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this thesis, we introduce different analyses for concurrent languages. Chapter 1 provides a global vision of all these analyses, relating them to the different chapters and their associated publications included in Appendix A.

Chapter 3 defines two new static analyses that can be applied to languages with explicit synchronization such as CSP. Both techniques are based on program slicing. In particular, we introduce a method to slice CSP specifications, in such a way that, given a CSP specification and a slicing criterion, we produce a slice such that (i) it is a subset of the specification (i.e., it is produced by deleting some parts of the original specification); (ii) it contains all the parts of the specification that must be executed (in any execution) before the slicing criterion (MEB analysis); and (iii) we can also produce an augmented slice that also contains those parts of the specification that could be executed (in some execution) before the slicing criterion (CEB analysis).

We have presented two algorithms to compute the MEB and CEB analyses based on a new data structure, the CSCFG, that has shown to be more precise than the previously used graph SCFG. A CSCFG is formed by the sequence of expressions that are evaluated during an execution. These expressions are conveniently connected to form a graph. In addition, the source position (in the specification) of each literal (i.e., events, operators and process names) is also included in the CSCFG. This is very useful because it provides the CSCFG with the ability to determine what parts of the source code have been

executed and in what order. The advantage of the CSCFG is that it cares about contexts, and thus it is able to distinguish between different contexts in which a process is called. This new data structure has been formalized and compared with the predecessor SCFG. Additionally, we introduce an algorithm to build the CSCFG associated with a CSP specification. The algorithm uses an instrumentation of the standard CSP's operational semantics to explore all possible computations of a specification. The semantics is deterministic because the rule applied in every step is predetermined by the initial state and the information in the stack. Therefore, the algorithm can execute the semantics several times to iteratively explore all computations and hence, generate the whole CSCFG. The CSCFG is generated even for non-terminating specifications due to the use of a loop detection mechanism controlled by the semantics. This semantics is an interesting result because it can serve as a reference to prove properties such as completeness of static analyses based on the CSCFG. The way in which the semantics has been instrumented can be used for other similar purposes with slight modifications.

On the practical side, we have implemented a tool called *SOC* [53] which is able to automatically generate the CSCFG of a CSP specification. It implements all the data structures and algorithms defined in Papers 1 and 2; and we have integrated it into the system ProB. *SOC* has been integrated into the most extended CSP animator and model-checker ProB [16, 49], that shows the maturity and usefulness of this tool and of CSCFGs. The last release of *SOC* implements the algorithm described in this chapter. However, in the implementation the algorithm is much more complex because it contains some improvements that significantly speed up the CSCFG construction. The implementation, source code and several examples are publicly available at [51]. Finally, a number of experiments conducted with SOC have been presented and discussed. These experiments demonstrated the usefulness of the technique for different applications such as debugging, program comprehension, program specialization and program simplification.

Chapter 4 introduces the first semantics of CSP instrumented for tracking. Therefore, it is an interesting result because it can serve as a reference mark to define and prove properties such as completeness of static analyses which are based on tracks. The execution of the tracking semantics produces a graph as a side effect which is the track of the computation. This track is produced step by step from the semantics, and thus, it can also be used to produce a track of an infinite computation until it is stopped. The generated track can be useful not only for tracking computations but for debugging and program comprehension. This is due to the fact that our generated

track also includes the specification positions associated with the expressions appearing in the track. Therefore, tracks could be used to analyse what parts of the program are executed (and in what order) in a particular computation. Also, this information allows a track viewer tool to highlight the parts of the code that are executed in each step. Notable analyses that use tracks are [23, 25, 24, 11]. The introduction of this semantics allows us to adapt these analyses to CSP. On the practical side, we have implemented a tool called CSP-Tracker [55] which is able to automatically generate tracks of a CSP specification.

Chapter 5 introduces an algorithm to automatically build a Petri net which produces the same sequences of observable events as a given CSP specification. The algorithm uses an instrumentation of the standard CSP's operational semantics to explore all possible computations of a specification. The semantics is deterministic because the rule applied in every step is predetermined by the initial configuration. Therefore, the algorithm can execute the semantics several times to iteratively explore all computations and hence, generate the whole Petri net. The Petri net is generated even for non-terminating specifications due to the use of a loop detection mechanism controlled by the semantics. This semantics is an interesting result because it explicitly relates the CSP model with the Petri net and the Petri net generated is very similar (structurally) to the CSP specification.

The Petri net generated is closely related to the CSP specification because all possible executions force tokens to follow the transitions in such a way that they reproduce the steps of the CSP semantics. This is very interesting compared to previous approaches where the relation between both models is hardly noticeable. The main cause of this important property is that part of the complexity needed to fill the gap between both models has been translated to the semantics (instead of translating it to the generated Petri net). Hence, an important application of these Petri nets is program comprehension.

However, if we are interested in a reduced version of the Petri net we can further transform it with a transformation defined to remove repeated or unnecessary parts. The resulting Petri nets obtained with this transformation are very compact and can be used to perform different Petri net analyses that can be transferred to the CSP specification. Both transformations have been proved correct and terminating.

On the practical side, we have implemented a tool called $CSP2PN$ which is able to automatically generate a Petri net equivalent to a CSP specification.

The interested reader is referred to [57] where the implementation, source code and several examples are publicly available.

Finally, in Chapter 6 we have introduced two different techniques for dynamic slicing of Petri nets. To the best of our knowledge, this is the first approach to the dynamic slicing for Petri nets. The first analysis takes into account the Petri net and an initial marking, but produces a slice w.r.t. any possibly firing sequence. The second analysis further reduces the computed slice by fixing a particular firing sequence. In general, our slices are smaller than previous (static) approaches where no initial marking nor firing sequence were considered.

7.2 Future Work

We plan to adapt some of the ideas presented in Chapter 3 to other concurrent languages, for instance Erlang. The presented approach can be easily adapted to other languages, as once the graph is built, the algorithms are independent of the language. Similarly, the tracking approach of Chapter 4 could be also adapted to other languages with event based semantics in order to ease the user to match parts of the program with its execution. Regarding work presented in Chapter 5, we plan to extend the set of CSP operators to include sequential composition, parameterized process calls, hiding and renaming. And also, we will study the failures and divergences models in addition to the traces model. Finally, respecting ideas presented in Chapter 6, we plan to carry on an experimental evaluation of our slicing techniques in order to test their viability in practice. We also find it useful to extend our slicing technique to other kinds of Petri nets (e.g., coloured Petri nets [42] and marked-controlled reconfigurable nets [54]).

Bibliography

[1] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[2] Alfred V. Aho and Jeffrey D. Ullman. *The theory of parsing, translation, and compiling*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1972.

[3] G. Almási and D. Padua. Majic: Compiling matlab for speed and responsiveness. In *ACM SIGPLAN Notices*, volume 37(5), pages 294–303. ACM, 2002.

[4] The ProB Animator and Model Checker. Available at http://www.stups.uni-duesseldorf.de/ProB.

[5] Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent programming in ERLANG*. Prentice Hall, 1993.

[6] A. Bell and B.R. Haverkort. Sequential and distributed model checking of Petri nets. *International Journal on Software Tools for Technology Transfer*, 7(1):43–60, 2005.

[7] Eike Best, Raymond Devillers, and Maciej Koutny. The box algebra=petri nets+process expressions. *Information and Computation*, 178(1):44 – 100, 2002.

[8] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks*, 14:25–59, 1987.

[9] W. N. Borst, V. V. Goldman, and J. A. van Hulzen. GENTRAN 90: a REDUCE package for the generation of Fortran 90 code. In *Proceedings of the international symposium on Symbolic and algebraic computation*, ISSAC '94, pages 45–51. ACM, 1994.

[10] Jeremy T. Bradley and William J. Knottenbelt. The ipc/HYDRA tool chain for the analysis of PEPA models. In *Proceedings of the First International Conference on the Quantitative Evaluation of Systems, 2004 (QEST 2004)*, pages 334–335. IEEE Computer Society, 2004.

[11] Bernd Brassel, Michael Hanus, Frank Huch, and Germán Vidal. A semantics for tracing declarative multi-paradigm programs. In *Proceedings of the 6th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP '04)*, pages 179–190. ACM, 2004.

[12] I. Brückner. Slicing CSP-OZ Specifications. In *Proceedings of the 16th Nordic Workshop on Programming Theory*, number 2004-041 in Technical Reports of the Department of Information Technology, pages 71–73. Uppsala University, Sweden, 2004.

[13] Ingo Brückner and Heike Wehrheim. Slicing an integrated formal method for verification. In *Proceedings of the 7th International Conference on Formal Engineering Methods (ICFEM 2005), Formal Methods and Software Engineering*, Lecture Notes in Computer Science, pages 360–374. Springer, 2005.

[14] Ingo Brückner and Heike Wehrheim. Slicing object-z specifications for verification. In *Proceedings of the 4th International Conference of B and Z Users (ZB 2005). Formal Specification and Development in Z and B*, volume 3455 of *Lecture Notes in Computer Science*, pages 414–433. Springer, 2005.

[15] Bettina Buth, Jan Peleska, and Hui Shi. Combining methods for the livelock analysis of a fault-tolerant system. In *Proceedings of the 7th International Conference on Algebraic Methodology and Software Technology (AMAST '98)*, volume 1548 of *Lecture Notes in Computer Science*, pages 124–139. Springer, 1999.

[16] Michael J. Butler and Michael Leuschel. Combining CSP and B for specification and property verification. In *Proceedings of the International Symposium of Formal Methods (FM 2005)*, volume 3582 of *Lecture Notes in Computer Science*, pages 221–236. Springer, 2005.

[17] David Callahan and Jaspal Sublok. Static analysis of low-level synchronization. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging*, PADD '88, pages 100–111. ACM, 1988.

[18] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theoretical computer science*, 240(1):177–213, 2000.

[19] C.K. Chang and H. Wang. A Slicing Algorithm of Concurrency Modeling Based on Petri Nets. In *Proceedings of the International Conference on Parallel Processing (ICPP'86)*, pages 789–792. IEEE Computer Society Press, 1986.

[20] J. Chang and D. Richardson. Static and dynamic specification slicing. In *Proceedings of the Fourth Irvine Software Symposium*. Irvine, CA, 1994.

[21] J. Cheng. Slicing Concurrent Programs - A Graph-Theoretical Approach. *Automated and Algorithmic Debugging*, pages 223–240, 1993.

[22] Derek Chiou. Using GCC as an Efficient, Portable Back-End. In *In Proceedings of the MIT Student Workshop for Scalable Computing*, pages 800–1, 1995.

[23] O. Chitil. A semantics for tracing. In *Draft Proceedings of the 13th International Workshop on Implementation of Functional Languages, IFL 2001*, pages 249–254, 2001.

[24] Olaf Chitil and Yong Luo. Structure and properties of traces for functional programs. *Electronic Notes in Theoretical Computer Science*, 176(1):39–63, 2007.

[25] Olaf Chitil, Colin Runciman, and Malcolm Wallace. Transforming Haskell for Tracing. In *Revised Selected Papers of the 14th International Workshop on Implementation of Functional Languages (IFL 2002)*, volume 2670 of *Lecture Notes in Computer Science*, pages 165–181. Springer, 2003.

[26] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, Cambridge, MA, 2000.

[27] F. de Cindio, G. de Michelis, L. Pomello, and C. Simone. A Petri Net Model of CSP. In *Proceedings of Convención Informática Latina (CIL'81)*, pages 392–406, Barcelona, 1981.

[28] P. Degano, R. Gorrieri, and S. Marchetti. An exercise in concurrency: A CSP process as a condition/event system. *Advances in Petri Nets 1988*, pages 85–105, 1988.

[29] J. Desel and J. Esparza. *Free choice Petri nets*. Cambridge University Press, New York, NY, USA, 1995.

[30] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[31] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of Computing*, STOC '78, pages 114–118. ACM, 1978.

[32] U. Goltz and W. Reisig. CSP-programs as nets with individual tokens. *Advances in Petri Nets 1984*, pages 169–196, 1985.

[33] Diganta Goswami and Rajib Mall. Fast slicing of concurrent programs. In *Proceedings of the 6th International Conference on High Performance Computing (HiPC'99)*, volume 1745 of *Lecture Notes in Computer Science*, pages 38–42. Springer, 1999.

[34] A. Hall and R. Chapman. Correctness by construction: Developing a commercial secure system. *IEEE Software*, 19(1):18–25, 2002.

[35] Mary Jean Harrold, Gregg Rothermel, and Saurabh Sinha. Computation of interprocedural control dependence. In *Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '98, pages 11–20. ACM, 1998.

[36] M.P.E. Heimdahl and M.W. Whalen. Reduction and Slicing of Hierarchical State Machines. In *Proceedings of the 6th European Software Engineering Conference (ESEC/FSE'97)*, pages 450–467. Springer LNCS 1301, 1997.

[37] Frank Heitmann, Daniel Moldt, Heiko Rölke, Kjeld H. Mortensen, Olaf Kummer, and Søren Christensen. Petri Nets Tool Database. Available at http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db.html.

[38] Jane Hillston. *A compositional approach to performance modelling*. Cambridge University Press, 1996.

[39] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[40] Gerard J. Holzmann. *Design and validation of computer protocols.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[41] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to automata theory, languages, and computation.* Addison-wesley, 2007.

[42] Kurt Jensen. Coloured petri nets. In *Proceedings of Advances in Petri Nets; Part I; Petri Nets: Central Models and Their Properties*, volume 254 of *Lecture Notes in Computer Science*, pages 248–299. Springer, 1987.

[43] S.P. Jones and A. Santos. Compilation by transformation in the glasgow haskell compiler. *Functional Programming, Glasgow*, pages 184–204, 1994.

[44] K.M. Kavi, F.T. Sheldon, B. Shirazi, and A.R. Hurson. Reliability analysis of CSP specifications using Petri nets and Markov processes. In *Proceedings of the Twenty-Eighth Hawaii International Conference on System Sciences*, volume 2, pages 516–524. IEEE, 1995.

[45] Jens Krinke. Static slicing of threaded programs. In *Proceedings of the SIGPLAN/SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (PASTE '98)*, pages 35–42. ACM, 1998.

[46] Jens Krinke. Context-sensitive slicing of concurrent programs. In *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference (ESEC/FSE 2003)*, pages 178–187. ACM, 2003.

[47] P.B. Ladkin and B.B. Simons. Static deadlock analysis for CSP-type communications. *Responsive Computer Systems: Steps Toward Fault-Tolerant Real-Time Systems*, pages 89–102, 1995.

[48] W.J. Lee, S.D. Cha, Y.R. Kwon, and H.N. Kim. A Slicing-based Approach to Enhance Petri Net Reachability Analysis. *Journal of Research and Practice in Information Technology*, 32(2):131–143, 2000.

[49] M. Leuschel and M. Butler. ProB: an automated analysis toolset for the B method. *International Journal on Software Tools for Technology Transfer (STTT)*, 10(2):185–203, 2008.

[50] M. Leuschel and M. Fontaine. Probing the depths of CSP-M: A new FDR-compliant validation tool. *Formal Methods and Software Engineering*, pages 278–297, 2008.

[51] Michael Leuschel, Marisa Llorens, Javier Oliver, Josep Silva, and Salvador Tamarit. The CSP's Slicer SOC. Available at http://users.dsic.upv.es/~jsilva/soc.

[52] Michael Leuschel, Marisa Llorens, Javier Oliver, Josep Silva, and Salvador Tamarit. The MEB and CEB Static Analysis for CSP Specifications. In *Revised Selected Papers of the 18th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2008)*, volume 5438 of *Lecture Notes in Computer Science*, pages 103–118. Springer, 2009.

[53] Michael Leuschel, Marisa Llorens, Javier Oliver, Josep Silva, and Salvador Tamarit. SOC: a slicer for CSP specifications. In *Proceedings of the 2009 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM 2009)*, pages 165–168. ACM, 2009.

[54] M. Llorens and J. Oliver. Introducing Structural Dynamic Changes in Petri Nets: Marked-Controlled Reconfigurable Nets. In *Proceedings of the 2nd International Conference on Automated Technology for Verification and Analysis (ATVA'04)*, pages 310–323. Springer LNCS 3299, 2004.

[55] Marisa Llorens, Javier Oliver, Josep Silva, and Salvador Tamarit. CSP-Tracker: Generator of CSP tracks. Available at https://github.com/mistupv/csp_tracker.

[56] Marisa Llorens, Javier Oliver, Josep Silva, and Salvador Tamarit. CSP-Tracker: Generator of CSP tracks. The web interface. Available at http://kaz.dsic.upv.es/csp_tracker.html.

[57] Marisa Llorens, Javier Oliver, Josep Silva, and Salvador Tamarit. CSP2PN: Translating CSP to Petri nets. Available at http://users.dsic.upv.es/~jsilva/CSP2PN/.

[58] Marisa Llorens, Javier Oliver, Josep Silva, and Salvador Tamarit. CSP2PN: Translating CSP to Petri nets. The web interface. Available at http://kaz.dsic.upv.es/csp2petri.html.

[59] Marisa Llorens, Javier Oliver, Josep Silva, and Salvador Tamarit. An algorithm to generate the context-sensitive synchronized control flow graph. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC 2010)*, pages 2144–2148. ACM, 2010.

[60] Marisa Llorens, Javier Oliver, Josep Silva, and Salvador Tamarit. Translating CSP Specifications to Equivalent Petri Nets. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2010)*, pages 320–326. CSREA Press, 2010.

[61] J. Magott. Performance evaluation of communicating sequential processes (CSP) using Petri nets. In *IEE Proceedings-e; Computers and Digital Techniques*, volume 139(3), pages 237–241. IET, 1992.

[62] A. Mazzeo, N. Mazzocca, S. Russo, C. Savy, and V. Vittorini. Formal specification of concurrent systems: a structured approach. *The Computer Journal*, 41(3):145–162, 1998.

[63] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and computation*, 100(1):1–77, 1992.

[64] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[65] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, (parts i and ii). *Information and computation*, 100(1):1–77, 1992.

[66] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[67] Mangala Gowri Nanda and S. Ramesh. Slicing concurrent programs. In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '00, pages 180–190. ACM, 2000.

[68] V. Natarajan and G.J. Holzmann. Outline for an operational semantics of PROMELA. *The SPIN Verification Systems. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. AMS*, 32:133–152, 1997.

[69] PIPE2: Platform Independent Petri net Editor 2. Available at http://pipe2.sourceforge.net/.

[70] Martin Odersky, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[71] E.R. Olderog. Operational Petri net semantics for CCSP. *Advances in Petri Nets 1987*, pages 196–223, 1987.

[72] J.L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[73] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Darmstadt University of Technology, Germany, 1962.

[74] A. Rakow. Slicing Petri Nets. Technical report, Department für Informatik, Carl von Ossietzky Universität, Oldenburg, 2007.

[75] A. Rakow. Slicing Petri Nets with an Application to Workflow Verification. In *Proceedings of the 34th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2008)*, pages 436–447. Springer LNCS 4910, 2008.

[76] A. Rakow. *Slicing and Reduction Techniques for Model Checking Petri Nets*. PhD thesis, Fakultät II Informatik, Wirtschafts- und Rechtswissenschaften Department für Informatik; Carl von Ossietzky Universität Oldenburg, 2011.

[77] M. Rauhamaa. *A Comparative Study of Methods for Efficient Reachability Analysis*. Licentiate's thesis, Helsinki University of Technology, Department of Computer Science and Engineering, Digital Systems Laboratory, 1990.

[78] The Petri Net Markup Language reference site. Available at http://www.pnml.org/.

[79] M. Ribaudo. Stochastic Petri net semantics for stochastic process algebras. In *Proceedings of the Sixth International Workshop on Petri Nets and Performance Models*, pages 148–157. IEEE, 1995.

[80] A. W. Roscoe, Paul H. B. Gardiner, Michael Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood. Hierarchical Compression for Model-Checking CSP or How to Check 10^{20} Dining Philosophers for Deadlock. In *Proceedings of the First International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS '95)*, volume 1019 of *Lecture Notes in Computer Science*, pages 133–152. Springer, 1995.

[81] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 2005.

[82] S. Rugaber. Program comprehension. *Encyclopedia of Computer Science and Technology*, 35(20):341–368, 1995.

[83] Frederick T. Sheldon. *Specification and analysis of stochastic properties for concurrent systems expressed using CSP*. PhD thesis, Computer Science and Engineering Deptment, 1996.

[84] Josep Silva. A vocabulary of program slicing-based techniques. *ACM Computing Surveys*, 44(3):12, 2012.

[85] A.M. Sloane and J. Holdsworth. Beyond traditional program slicing. In *Proc. of the Int'l Symp. on Software Testing and Analysis*, pages 180–186. ACM Press, 1996.

[86] F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[87] E. Visser. A survey of rewriting strategies in program transformation systems. *Electronic Notes in Theoretical Computer Science*, 57:109–143, 2001.

[88] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[89] J. Zhao. Slicing aspect-oriented software. In *Proceedings of the 10th International Workshop on Program Comprehension*, pages 251–260. IEEE, 2002.

[90] J. Zhao, J. Cheng, and K. Ushijima. Slicing concurrent logic programs. In *Proceedings of the 2nd Fuji International Workshop on Functional and Logic Programming*, pages 143–162, 1997.

Appendix A

Papers of the Thesis

List of papers:

1. Michael Leuschel, Marisa Llorens, Javier Oliver, Josep Silva, and Salvador Tamarit. Static slicing of explicitly synchronized languages. *Information and Computation*, 214:10–46, 2012.

2. Marisa Llorens, Javier Oliver, Josep Silva, and Salvador Tamarit. Graph generation to statically represent CSP processes. In *Revised Selected Papers of the 18th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2008)*, volume 6564 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 2010.

3. Marisa Llorens, Javier Oliver, Josep Silva, and Salvador Tamarit. A tracking semantics for CSP. In *Proceedings of the 10th International Conference on Mathematics of Program Construction (MPC 2010)*, volume 6120 of *Lecture Notes in Computer Science*, pages 248–270. Springer, 2010.

4. M. Llorens, J. Oliver, J. Silva, and S. Tamarit. Generating a Petri net from a CSP specification: A semantics-based method. *Advances in Engineering Software*, 2012.

5. Marisa Llorens, Javier Oliver, Josep Silva, Salvador Tamarit, and Germán Vidal. Dynamic slicing techniques for Petri nets. *Electronic Notes in Theoretical Computer Science*, 223:153–165, 2008.

The personal contributions of the author in these papers are the following:

1. Collaboration in the definition of the CSCFG and the algorithms presented. Implementation of the slicer, and integration in ProB. Definition of some formal aspects. Proof of some theorems and lemmas.

2. Definition and implementation of the main algorithm and its instrumented operational semantics. Definition of some formal aspects. Proof of some theorems and lemmas.

3. Definition and implementation of the main algorithm and its instrumented operational semantics. Definition of some formal aspects. Proof of some theorems and lemmas.

4. Definition and implementation of the algorithm and the instrumented operational semantics. Improvements in the resulting nets. Definition of some formal aspects. Proof of some theorems and lemmas.

5. Collaboration in the definition of the algorithms. Definition of some formal aspects. Proof of some theorems and lemmas.

Other publications of the author related with this thesis are [52, 53, 60, 59], but those works are somehow subsumed by the papers considered here. Additionally, the author have other important publications not included in this thesis because they were not related to the main topic of the thesis. The most relevant are:

- Sebastian Fischer, Josep Silva, Salvador Tamarit, and Germán Vidal. Preserving sharing in the partial evaluation of lazy functional programs. *In Revised Selected Papers of the 17th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2007)*, volume 4915 of *Lecture Notes in Computer Science*, pages 74–89. Springer, 2007.

- Gustavo Arroyo, J. Guadalupe Ramos, Salvador Tamarit, and Germán Vidal. A transformational approach to polyvariant BTA of higher-order functional programs. *Revised Selected Papers of the 18th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2008)*, volume 5438 of *Lecture Notes in Computer Science*, pages 40–54. Springer, 2008.

- Michael Leuschel, Salvador Tamarit, and Germán Vidal. Fast and accurate strong termination analysis with an application to partial evaluation. In *Revised Selected Papers of the 18th International Workshop on*

Functional and Constraint Logic Programming (WFLP 2009), volume 5979 of *Lecture Notes in Computer Science*, pages 111–127. Springer, 2009.

- Jesús Manuel Almendros-Jiménez, Josep Silva, and Salvador Tamarit. XQuery optimization based on program slicing. In *Proceedings of the 20th ACM Conference on Information and Knowledge Management (CIKM 2011)*, pages 1525–1534. ACM, 2011.

- Josep Silva, Salvador Tamarit, and César Tomás. System dependence graphs in sequential Erlang. In *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering, (FASE 2012)*, volume 7212 of *Lecture Notes in Computer Science*, pages 486–500. Springer, 2012.

- Konstantinos F. Sagonas, Josep Silva, and Salvador Tamarit. Precise explanation of success typing errors. In *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation, (PEPM 2013), Rome, Italy, January 21-22, 2013*, pages 33–42. ACM, 2013.

- David Insa, Josep Silva, and Salvador Tamarit. Using the words/leafs ratio in the DOM tree for content extraction. *The Journal of Logic and Algebraic Programming (JLAP)*, Publication pending, 2013.

Static Slicing of Explicitly Synchronized Languages☆

Michael Leuschel[a], Marisa Llorens[b], Javier Oliver[b], Josep Silva*,[b], Salvador Tamarit[b]

[a] *Institut für Informatik, Heinrich-Heine-Universität Düsseldorf, Universitätsstrasse 1, D-40225 Düsseldorf, Germany*
[b] *Universitat Politècnica de València, Camino de Vera S/N, E-46022 Valencia, Spain*

Abstract

Static analysis of concurrent languages is a complex task due to the non-deterministic execution of processes. If the concurrent language being studied allows process synchronization, then the analyses are even more complex (and thus expensive), e.g., due to the phenomenon of *deadlock*. In this work we introduce a static analysis technique based on program slicing for concurrent and explicitly synchronized languages in general, and CSP in particular. Concretely, given a particular point in a specification, our technique allows us to know what parts of the specification must necessarily be executed before this point, and what parts of the specification could be executed before it. Our technique is based on a new data structure that extends the *Synchronized Control Flow Graph* (SCFG). We show that this new data structure improves the SCFG by taking into account the context in which processes are called and, thus, it makes the slicing process more precise. The technique has been implemented and tested with real specifications, producing good results. After formally defining our technique, we describe our tool, its architecture, its main applications and the results obtained from several experiments conducted in order to measure the performance of the tool.

Key words: Concurrent Programming, CSP, Program Slicing

1. Introduction

Process algebras such as CSP [9], π-calculus [19] or LOTOS [1] and process modeling languages such as Promela [10, 21] allow us to specify complex systems with multiple interacting processes. The study and transformation of such

☆This work has been partially supported by the Spanish *Ministerio de Economía y Competitividad (Secretaría de Estado de Investigación, Desarrollo e Innovación)* under grant TIN2008-06622-C03-02 and by the *Generalitat Valenciana* under grant PROMETEO/2011/052. Salvador Tamarit was partially supported by the Spanish MICINN under FPI grant BES-2009-015019.

*Corresponding author

Email addresses: leuschel@cs.uni-duesseldorf.de (Michael Leuschel), mllorens@dsic.upv.es (Marisa Llorens), fjoliver@dsic.upv.es (Javier Oliver), jsilva@dsic.upv.es (Josep Silva), stamarit@dsic.upv.es (Salvador Tamarit)

systems often implies different analyses (e.g., deadlock analysis [14], reliability analysis [11], refinement checking [24], etc.).

In this work we introduce a static analysis technique for process algebras with explicit synchronization mechanisms, based on a well-known program comprehension technique called *program slicing* [26]. Program slicing is a method for decomposing programs by analyzing their data and control flow. Roughly speaking, a *program slice* consists of those parts of a program that are (potentially) determining the values computed at some program point and/or variable, referred to as a *slicing criterion*. Program slices are usually computed from a *Program Dependence Graph* (PDG) [6] that makes explicit both the data and control dependences for each operation in a program. Program dependences can be traversed backwards or forwards (from the slicing criterion), that is known as *backward* or *forward* slicing, respectively. Additionally, slices can be *dynamic* or *static*, depending on whether a concrete program's input is provided or not. A survey on program slicing can be found, e.g., in [25].

Our technique allows us to extract the part of a specification related to a given point (referred to as the slicing criterion) in the specification. This technique can be very useful to debug, understand, maintain and reuse specifications; but also as a preprocessing stage of other analyses and/or transformations in order to reduce the complexity of the specification. In particular, given a point (e.g., an event) in a specification, our technique allows us to extract those parts of the specification that must be executed before the specified point (thus they are an implicit precondition); and those parts of the specification that could be executed before it. Therefore, the other parts of the specification cannot be executed before this point.

Example 1. *Consider the following specification*[1]*:*

```
MAIN = (STUDENT ||{pass} PARENT) ||{pass,fail} COLLEGE
STUDENT = year1 → (pass → YEAR2 □ fail → STUDENT)
YEAR2 = year2 → (pass → YEAR3 □ fail → YEAR2)
YEAR3 = year3 → (pass → graduate → STOP □ fail → YEAR3)
PARENT = pass → present → PARENT
COLLEGE = fail → COLLEGE □ pass → C1
C1 = fail → COLLEGE □ pass → C2
C2 = fail → COLLEGE □ pass → prize → STOP
```

In this specification we have three processes (`STUDENT`*,* `PARENT` *and* `COLLEGE`*) executed in parallel and synchronized on common events. Process* `STUDENT` *represents the three-year academic courses of a student; process* `PARENT` *represents the parent of the student who gives her a present when she passes a course; and*

[1] In the following, without lack of generality, we will use the *Communicating Sequential Processes* (CSP) [9] language as the running language for our examples. We refer those readers non familiar with CSP syntax to Section 2 where we provide a brief introduction to CSP.

process `COLLEGE` *represents the college who gives a prize to those students that finish without any fail.*

We are interested in determining what parts of the specification must be executed before the student fails in the second year, hence, we mark event `fail` *of process* `YEAR2` *(thus the slicing criterion is* `(YEAR2, fail)`*, marked by a box in the above figure). Our slicing technique automatically extracts the slice consisting of the expressions in black. We can additionally be interested in knowing what parts could be executed before the same event. In this case, our technique adds to the slice the underscored parts because they could be executed (in some executions) before the marked event (observe that the result of this analysis is always a superset of the result obtained by the previous analysis). Therefore, this analysis could be used for program comprehension. Note, for instance, that in order to fail in the second year, the student has necessarily passed the first year. But, the parent may or may not have given a present to his daughter (even if she passed the first year) because this specification does not force the parent to give a present to his daughter until she has passed the second year. Moreover, note that the choice of process* `C1` *belongs also to the slice. This is due to the fact that the slicing criterion must synchronize with the event* `fail` *of this process; therefore, the choice must be executed before the slicing criterion.*[2] *This is not so obvious from the specification, and the slice can help to understand the actual meaning of the specification.*

Computing the parts of the specification that could be executed before the slicing criterion can be useful, e.g., for debugging. If the slicing criterion is an event that executed incorrectly (i.e., it should not happen in the execution), then the slice produced contains all the parts of the specification that could produce the wrong behavior.

A third application is program specialization. Note that the slices produced are not executable, but, in both cases, the slices could be made executable by replacing the removed parts by "`STOP`*" or by "*$\rightarrow$ `STOP`*" if the removed expression has a prefix. Hence, we have defined a further transformation that allows us to extract executable slices. The specialized specification contains all the necessary parts of the original specification whose execution leads to the slicing criterion (and then, the specialized specification finishes).*

We have implemented our technique producing the first program slicer for CSP specifications. In our implementation, the slicing process is completely automatic. Once the user has loaded a specification, she can select (with the mouse) the point she is interested in. Obviously, this simple action is enough to define a slicing criterion because the tool can automatically determine the process and the source position of interest. This implementation is a tool that has been integrated in the system ProB [15, 3], an animator and model checker for B and CSP. We will describe this tool in Section 5.

It should be clear that computing the minimum slice of an arbitrary CSP specification is an undecidable problem. Consider for instance the following CSP specification:

[2] We could have chosen also to include the `fail` event of `C1` into the slice. This is a matter of taste.

```
MAIN = P ⊓ Q

P = X ; Q

Q = a → STOP

X = Infinite Process
```

together with the slicing criterion (`Q`, `a`). Determining whether `X` does not belong to the slice implies determining whether `X` terminates, which is undecidable.

The main contributions of this work are the following:

- We define two new static analyses for process algebras and propose algorithms for their implementation. Despite their clear usefulness we have not found similar static analyses in the literature.
- We define the *context-sensitive synchronized control flow graph* and show its advantages over its predecessors. This is a new data structure able to represent all computations of a specification taking into account the context of process calls; and it is particularly interesting for slicing languages with explicit synchronization.
- We have implemented our technique and integrated it in ProB [15, 3, 16]. Current releases of ProB are distributed with the slicer as an analysis tool. We present the implementation and the results obtained with several benchmarks.

The rest of the paper is organized as follows. In Section 2 we give an overview of the syntax and semantics of a process algebra (CSP) and introduce some notation that will be used along the article. In this section we also introduce an extension of the standard operational semantics of CSP. In Section 3 we show that previous data structures used in program slicing are inaccurate or inappropriate in our context, and we introduce the *Context-sensitive Synchronized Control Flow Graph* (CSCFG) as a solution and discuss its advantages over its predecessors. Our slicing technique is presented in Section 4 where we introduce two algorithms to slice CSP specifications from their CSCFGs. In Section 5 we present our implementation, we describe the architecture of our tool SOC, and we show the results of some experiments that reflect the efficiency and performance of the tool. Next, we discuss some related work in Section 6 and, finally, Section 7 concludes. All proofs of technical results can be found in Appendix A.

2. Communicating Sequential Processes

In order to keep the paper self-contained, in this section we recall the syntax and the semantics of the constructs used in our process algebra specifications. We use the CSP language [9], but the concepts and algorithms can also be applied to other process algebras. We also introduce here some notation that will be used along the paper.

Figure 1 summarizes the syntax constructions used in our CSP specifications. More precisely, a specification $\mathcal{S}$ is a finite collection of definitions. The left-hand side of each definition is the name of a different process, that is defined

$\mathcal{S}$	::=	$D_1 \dots D_m$	(entire specification)	*Domains* $M, N, O \dots \in \mathcal{N}ames$ (names) $P, Q, R \dots \in \mathcal{P}$ (processes) $a, b, c \dots \in \Sigma$ (events) $u, v, w \dots \in \mathcal{V}$ (variables)
D	::=	$M = P$	(process definition)	
	|	$M(\overline{x_n}) = P$	(parameterized process)	where $\overline{x_n} = x_1, \dots, x_n$ and $x_i \in \Sigma \cup \mathcal{V}$
P	::=	M	(process call)	
	|	$M(\overline{x_n})$	(parameterized process call)	
	|	$x \rightarrow P$	(prefixing)	
	|	$c?u \rightarrow P$	(input)	
	|	$c!u \rightarrow P$	(output)	
	|	$P \sqcap Q$	(internal choice)	
	|	$P \,\Box\, Q$	(external choice)	
	|	$P \lessdot bool \gtrdot Q$	(conditional choice)	where $bool \in \{true, false\}$
	|	$P \mid\mid\mid Q$	(interleaving)	
	|	$P \underset{X}{\parallel} Q$	(synchronized parallelism)	where $X \subseteq \Sigma \cup \mathcal{V}$
	|	$P \,;\, Q$	(sequential composition)	
	|	$P \backslash X$	(hiding)	
	|	$P[\![f]\!]$	(renaming)	where $f : (\Sigma \cup \mathcal{V}) \rightarrow (\Sigma \cup \mathcal{V})$
	|	$SKIP$	(skip)	
	|	$STOP$	(stop)	

Figure 1: Syntax of CSP specifications

in the right-hand side (*rhs*) by means of an expression[3] that can be a call to another process or a combination of the following operators:

Prefixing. It specifies that event x (called the prefix) must happen before P.

Input. It is used to receive a message from another process. Message u is received through channel c; then process P is executed.

Output. It is analogous to the input, but this is used to send messages. Message u is sent through channel c; then process P is executed.

Internal choice. The system chooses (e.g., non-deterministically) to execute one of the two expressions.

External choice. It is identical to internal choice but the choice comes from outside the system (e.g., the user).

Conditional choice. It is a choice that depends on a condition, i.e., it is equivalent to `if` *bool* `then` P `else` Q.

Interleaving. Both expressions are executed in parallel and independently.

Synchronized parallelism. Both expressions are executed in parallel with a set of synchronized events. In absence of synchronization both expressions can execute in any order. Whenever a synchronized event $x_i, 1 \leqslant i \leqslant n$, happens in one of the expressions it must also happen in the other at the

[3]Therefore a process is defined by an expression, and thus, we often use indistinguishably these terms.

same time. Whenever the set of synchronized events is not specified, it is assumed that the expressions are synchronized in all common events.

Sequential composition. It specifies a sequence of two processes. When the first (successfully) finishes, the second starts.

Hiding. Process P is executed with a set of hidden events $\{\overline{x_n}\}$. Hidden events are not observable from outside the process, and thus, they cannot synchronize with other processes.

Renaming. Process P is executed with a set of renamed events specified with the total mapping f. An event a renamed as b behaves internally as a but it is observable as b from outside the process.

Skip. It finishes the current process. It allows the next sequential process to continue.

Stop. It finishes the current process; but it does not allow the next sequential process to continue.

Figure 2 shows the standard operational semantics of CSP as defined by A. W. Roscoe [23]. This semantics is a logical inference system where a state is formed by a single expression called the *control*. The system starts with an initial state, and the rules of the semantics are used to infer how this state evolves. When no rules can be applied to the current state, the computation finishes. The rules of the semantics change the states of the computation due to the occurrence of events. The set of possible events is $\Sigma \cup \{\tau, \checkmark\}$. Events in $\Sigma = \{a, b, c \ldots\}$ are visible from the external environment, and can only happen with its cooperation (e.g., actions of the user). The special event τ cannot be observed from outside the system and it happens automatically as defined by the semantics. $\checkmark$ is a special event representing the successful termination of a process. The special symbol $\top$ is used to denote any process that already terminated.

The intuitive meaning of each rule is the following:

((Parameterized) Process Call) The call is unfolded and the right-hand side of process M is added to the control.

(Prefixing) When event a occurs, process P is added to the control. This rule is used both for prefixing and communication operators (input and output). Given a communication expression, either $c?u \rightarrow P$ or $c!u \rightarrow P$, this rule treats the expression as a prefixing except for the fact that the set of messages appearing in P is replaced by the communicated events.

(SKIP) After `SKIP`, the only possible event is $\checkmark$, that denotes the end of the (sub)com-putation with the special symbol $\top$. There is no rule for $\top$ (nor for `STOP`), hence, this (sub)computation has finished.

(Internal Choice 1 and 2) The system uses the internal event τ to (non-deterministi-cally) select one of the two processes P or Q that is added to the control.

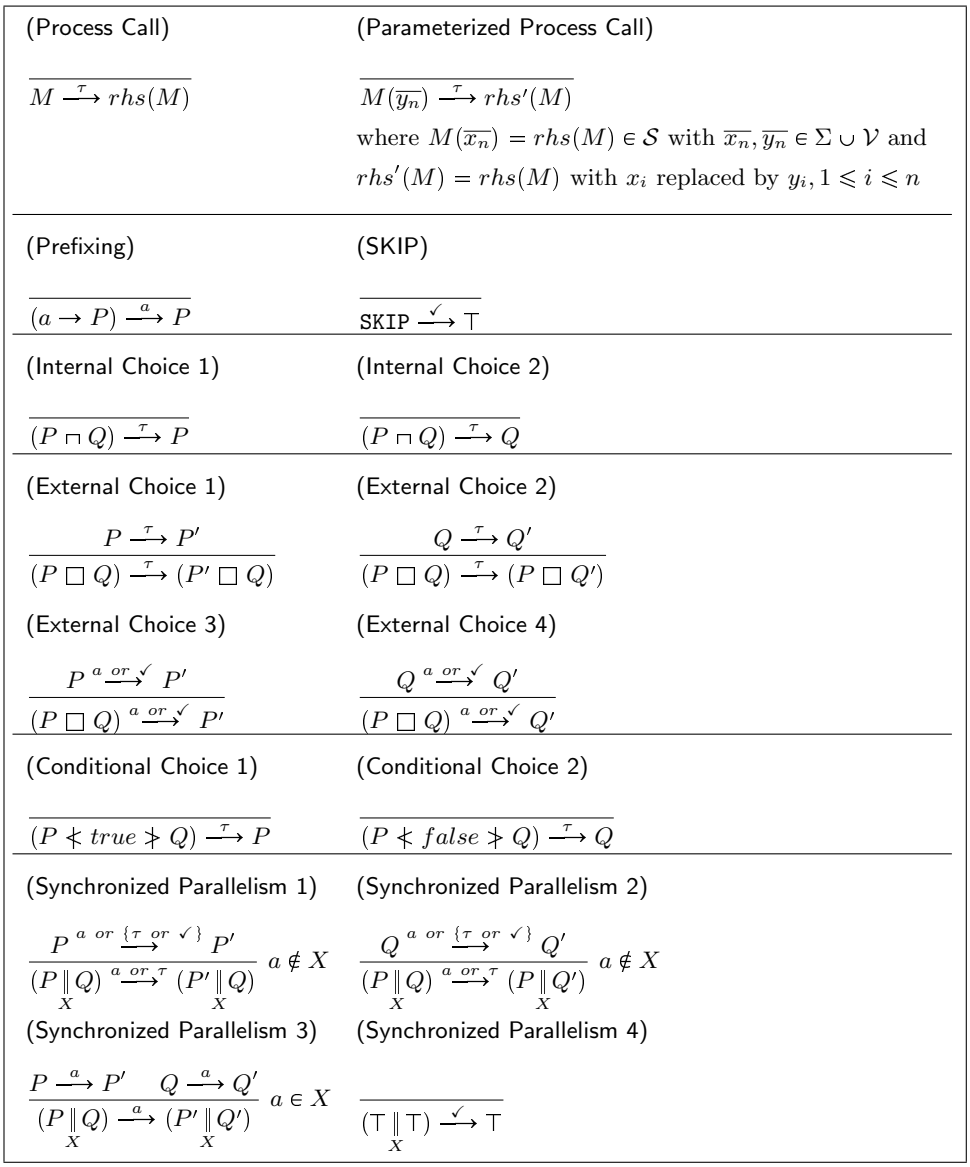


Figure 2: CSP's operational semantics

(External Choice 1, 2, 3 and 4) The occurrence of τ develops one of the processes. The occurrence of an event $a \neq \tau$ is used to select one of the two processes P or Q (the other process is discarded) and the control changes according to the event.

(Conditional Choice 1 and 2) The condition *bool* is evaluated. If it is $true$, process P is put in the control, if it is $false$, process Q is.

(Synchronized Parallelism 1 and 2) When event $a \notin X$ or events τ or $\checkmark$ happen, one of the two processes P or Q evolves accordingly, but only a is visible from outside the parallelism operator.

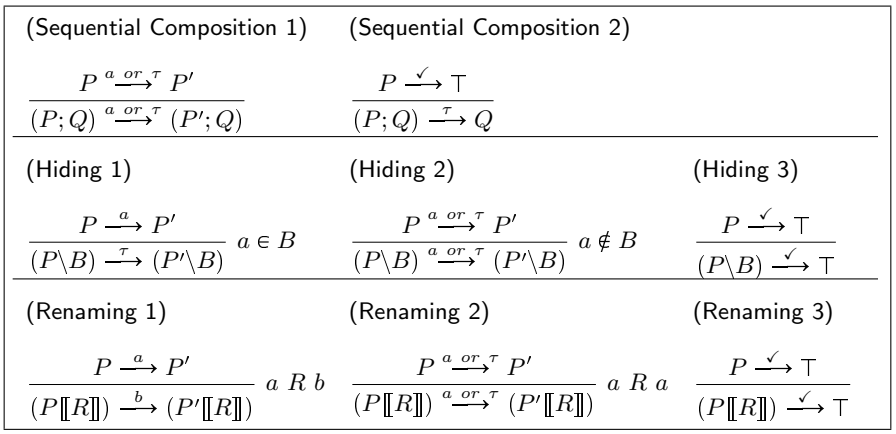


Figure 2: CSP's operational semantics (cont.)

(Synchronized Parallelism 3) When event $a \in X$ happens, it is required that both processes synchronize, P and Q are executed at the same time and the control becomes $P' \underset{X}{\parallel} Q'$.

(Synchronized Parallelism 4) When both processes have successfully terminated the control becomes $\top$, performing $\checkmark$.

(Sequential Composition 1) In $P;Q$, P can evolve to P' with any event except $\checkmark$. Hence, the control becomes $P';Q$.

(Sequential Composition 2) When P finishes (with event $\checkmark$), Q starts. Note that $\checkmark$ is hidden from outside the whole process becoming τ.

(Hiding 1) When event $a \in B$ occurs in P, it is hidden, and thus changed to τ so that it is not observable from outside P.

(Hiding 2 and Hiding 3) P can normally evolve (using rule 2) until it is finished ($\checkmark$ happens). When P finishes, rule 3 is used and the control becomes $\top$.

(Renaming 1) Whenever an event a happens in P, it is renamed to b ($a\ R\ b$) so that, externally, only b is visible.

(Renaming 2 and 3) Renaming has no effect on either events renamed to themselves ($a\ R\ a$), and τ or $\checkmark$ events. The rules for renaming are similar to those for hiding.

We illustrate the semantics with the following example.

Example 2. *Consider the following CSP specification:*

```
MAIN = (a → STOP) ||{a} (P □ (a → STOP))
P = b → SKIP
```

If we use rhs(`MAIN`) *as the initial state to execute the semantics, we get the computation (i.e., sequence of valid state transitions) shown in Figure 3 where*

the final state is $((\mathtt{a} \rightarrow \mathtt{STOP}) \underset{\{\mathtt{a}\}}{\parallel} \top)$. *This computation corresponds to the execution of the left branch of the choice (i.e.,* P*) and thus only event* b *occurs. Each rewriting step is labeled with the applied rule.*

$$\text{(Synchronized Parallelism 2)}\ \frac{\text{(External Choice 1)}\ \dfrac{\text{(Process Call)}\ \dfrac{}{\mathtt{P} \xrightarrow{\tau} (\mathtt{b} \rightarrow \mathtt{SKIP})}}{(\mathtt{P} \Box (\mathtt{a} \rightarrow \mathtt{STOP})) \xrightarrow{\tau} ((\mathtt{b} \rightarrow \mathtt{SKIP}) \Box (\mathtt{a} \rightarrow \mathtt{STOP}))}}{((\mathtt{a} \rightarrow \mathtt{STOP}) \underset{\{\mathtt{a}\}}{\parallel} (\mathtt{P} \Box (\mathtt{a} \rightarrow \mathtt{STOP}))) \xrightarrow{\tau} State_1}$$

$$\text{where } State_1 = ((\mathtt{a} \rightarrow \mathtt{STOP}) \underset{\{\mathtt{a}\}}{\parallel} ((\mathtt{b} \rightarrow \mathtt{SKIP}) \Box (\mathtt{a} \rightarrow \mathtt{STOP})))$$

$$\text{(Synchronized Parallelism 2)}\ \frac{\text{(External Choice 1)}\ \dfrac{\text{(Prefixing)}\ \dfrac{}{(\mathtt{b} \rightarrow \mathtt{SKIP}) \xrightarrow{\mathtt{b}} \mathtt{SKIP}}}{((\mathtt{b} \rightarrow \mathtt{SKIP}) \Box (\mathtt{a} \rightarrow \mathtt{STOP})) \xrightarrow{\mathtt{b}} \mathtt{SKIP}}}{State_1 \xrightarrow{\mathtt{b}} ((\mathtt{a} \rightarrow \mathtt{STOP}) \underset{\{\mathtt{a}\}}{\parallel} \mathtt{SKIP})}$$

$$\text{(Synchronized Parallelism 2)}\ \frac{\text{(SKIP)}\ \dfrac{}{\mathtt{SKIP} \xrightarrow{\checkmark} \top}}{((\mathtt{a} \rightarrow \mathtt{STOP}) \underset{\{\mathtt{a}\}}{\parallel} \mathtt{SKIP}) \xrightarrow{\tau} ((\mathtt{a} \rightarrow \mathtt{STOP}) \underset{\{\mathtt{a}\}}{\parallel} \top)}$$

Figure 3: A computation with the operational semantics in Figure 2

We need to define the notion of *specification position* that, roughly speaking, is a label that identifies a part of the specification. Formally,

Definition 1. (Position, Specification Position) Given a CSP specification $\mathcal{S}$ and a process definition $M = P$ in $\mathcal{S}$, the *positions* in each P are represented by a sequence of natural numbers, where Λ denotes the empty sequence (i.e., the root position). They are used to address the literals of an expression viewed as a tree, and are inductively defined as follows:

$P|_\Lambda = P$
$(P\ op)|_{1.w} = P|_w \quad \forall\ op \in \{\backslash, [\![]\!]\}$
$(P\ op\ Q)|_{1.w} = P|_w \quad \forall\ op \in \{\rightarrow, \sqcap, \Box, \not\leftrightarrow, |||, ||, ;\}$
$(P\ op\ Q)|_{2.w} = Q|_w \quad \forall\ op \in \{\rightarrow, \sqcap, \Box, \not\leftrightarrow, |||, ||, ;\}$

$P|_w$ is undefined otherwise.

A *specification position* is a pair (M, w) with $M = P \in \mathcal{S}$ and w a sequence of naturals, such that $P|_w$ is defined. We use the special specification position $(M, 0)$ for the left-hand side of the process definition $M = P$. We let $\mathcal{P}os(\mathcal{S})$ denote the set of all specification positions for processes in $\mathcal{S}$.

In the following we will refer to the literal associated to a specification position α with $lit(\alpha)$. For instance, in the specification of Example 3 where expressions are labeled with their associated specification positions, $lit((\mathtt{MAIN}, 1)) = ||$ and $lit((\mathtt{MAIN}, 1.1)) = \mathtt{BUS}$. As we will work with graphs whose nodes are labeled with specification positions, we often use $\mathcal{P}os(N)$ to denote the set of all specification positions associated with the set of nodes N.

Example 3. *In the following specification*[4] $\mathcal{S}$ *each expression has been labeled (in grey color) with its associated specification position so that all labels are unique.*

$$\texttt{MAIN}_{(\texttt{MAIN},0)} = (\texttt{BUS}_{(\texttt{MAIN},1.1)} \,||_{(\texttt{MAIN},1)}\, \texttt{P1}_{(\texttt{MAIN},1.2)})\,;_{(\texttt{MAIN},\Lambda)} \\ (\texttt{BUS}_{(\texttt{MAIN},2.1)} \,||_{(\texttt{MAIN},2)}\, \texttt{P2}_{(\texttt{MAIN},2.2)})$$

$$\texttt{BUS}_{(\texttt{BUS},0)} = \texttt{board}_{(\texttt{BUS},1)} \rightarrow_{(\texttt{BUS},\Lambda)} \texttt{alight}_{(\texttt{BUS},2.1)} \rightarrow_{(\texttt{BUS},2)} \texttt{SKIP}_{(\texttt{BUS},2.2)}$$

$$\texttt{P1}_{(\texttt{P1},0)} = \texttt{wait}_{(\texttt{P1},1)} \rightarrow_{(\texttt{P1},\Lambda)} \texttt{board}_{(\texttt{P1},2.1)} \rightarrow_{(\texttt{P1},2)} \\ \texttt{alight}_{(\texttt{P1},2.2.1)} \rightarrow_{(\texttt{P1},2.2)} \texttt{SKIP}_{(\texttt{P1},2.2.2)}$$

$$\texttt{P2}_{(\texttt{P2},0)} = \texttt{wait}_{(\texttt{P2},1)} \rightarrow_{(\texttt{P2},\Lambda)} \texttt{board}_{(\texttt{P2},2.1)} \rightarrow_{(\texttt{P2},2)} \texttt{pay}_{(\texttt{P2},2.2.1)} \rightarrow_{(\texttt{P2},2.2)} \\ \texttt{alight}_{(\texttt{P2},2.2.2.1)} \rightarrow_{(\texttt{P2},2.2.2)} \texttt{SKIP}_{(\texttt{P2},2.2.2.2)}$$

The notion of specification position allows us to determine what parts of the specification are executed in a particular execution. For this purpose, we have extended the semantics of Figure 2 in such a way that given a specification $\mathcal{S}$ and an execution of $\mathcal{S}$ with the extended semantics, the semantics produces as a side-effect the collection of specification positions that have been executed in this particular execution.

The extended semantics is presented in Figure 4 where we assume that every expression in the program has been labeled with its specification position (denoted by a subscript, e.g., P_α). A *state* of the semantics is a tuple (P, ω) where P is the *control*, i.e., the expression to be evaluated and ω represents the set of specification positions already evaluated. When the computation has finished or interrupted, ω contains the portion of the source code that has been executed.

An explanation for each rule of the semantics follows:

((Parameterized) Process Call) The called process is unfolded and its specification position α is added to the current set of specification positions ω. The new expression in the control is $rhs(M)$.

(Prefixing) Set ω is increased with the specification positions of the prefix and the prefixing operator.

(SKIP and STOP) The specification position α of `SKIP` (respectively `STOP`) is added to the current set of specification positions.

(Internal Choice 1 and 2) (Conditional Choice 1 and 2) The choice operator is added to ω.

(External Choice 1, 2, 3 and 4) External choices can develop both branches while τ events happen (rules 1 and 2), until an event in $\Sigma \cup \{\checkmark\}$ occurs (rules 3 and 4). This means that the semantics can develop both branches of the trace alternatively before selecting one branch. Of course, we want the extended semantics to collect all specification positions that have been

[4] This is a simplification of a benchmark by Simon Gay to simulate a bus line.

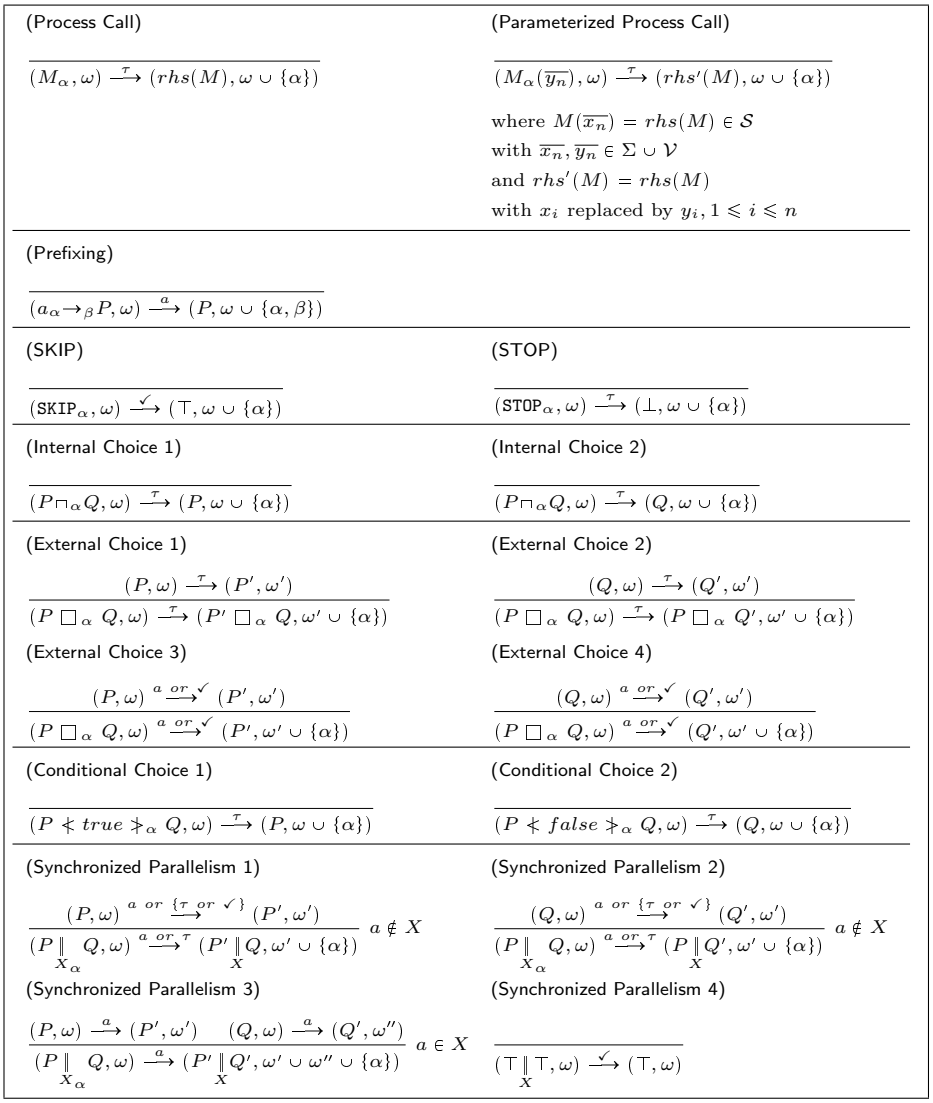


Figure 4: An instrumented operational semantics for CSP with specification positions

executed and thus, when rules 1 and 2 are fired several times to evolve the branches of the choice, the corresponding specification positions are added to the common set ω.

(Synchronized Parallelism 1 and 2) Because nodes from both parallel processes can be executed interweaved, the parallelism operator is added to ω together with the specification positions (ω') executed of the corresponding branch.

(Synchronized Parallelism 3) When a synchronization occurs, the parallelism operator together with the specification positions executed in both branches are added to ω.

(Synchronized Parallelism 4) It has no influence over the set ω because

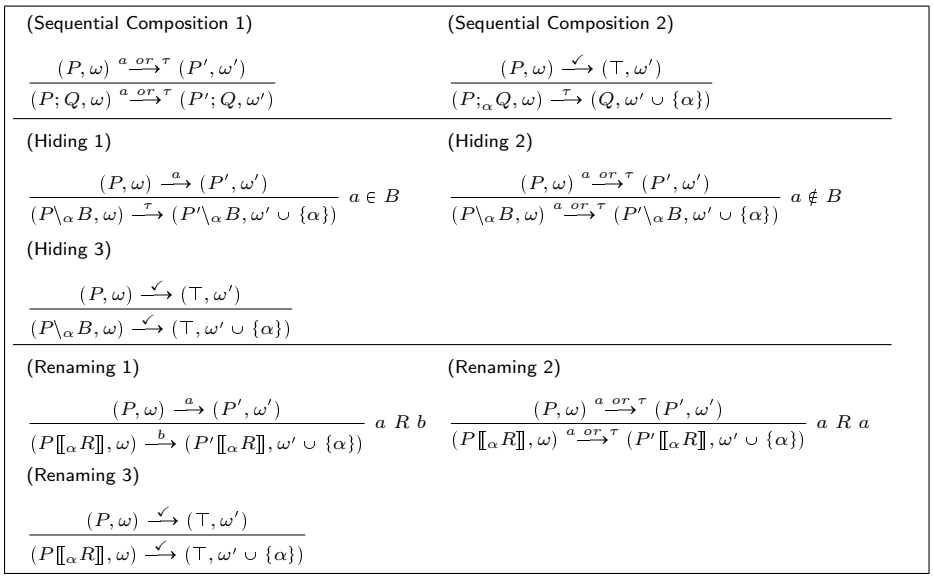


Figure 4: An instrumented operational semantics for CSP with specification positions (cont.)

the processes already terminated, and thus, the parallelism operator is already included in the set by the other rules.

(Sequential Composition 1 and 2) Sequential Composition 1 is used to add to ω the specification positions executed in process P until it is finished. When P finishes Sequential Composition 2 is used and the specification position of ; is added to ω.

(Hiding 1, 2 and 3) ω is increased with the specification position of the Hiding operator and the specification positions of the developed process P.

(Renaming 1, 2 and 3) It is completely analogous to the previous case.

Example 4. *Consider again the specification of Example 2 but now expressions are labeled with their associated specification positions (in grey color) so that labels are unique.*

$$\mathtt{MAIN}_{(\mathtt{MAIN},0)} = (\mathtt{a}_{(\mathtt{MAIN},1.1)} \rightarrow_{(\mathtt{MAIN},1)} \mathtt{STOP}_{(\mathtt{MAIN},1.2)}) \underset{\{\mathtt{a}\}}{\parallel}_{(\mathtt{MAIN},\Lambda)} (\mathtt{P}_{(\mathtt{MAIN},2.1)} \Box_{(\mathtt{MAIN},2)} (\mathtt{a}_{(\mathtt{MAIN},2.2.1)} \rightarrow_{(\mathtt{MAIN},2.2)} \mathtt{STOP}_{(\mathtt{MAIN},2.2.2)}))$$

$$\mathtt{P}_{(\mathtt{P},0)} = \mathtt{b}_{(\mathtt{P},1)} \rightarrow_{(\mathtt{P},\Lambda)} \mathtt{SKIP}_{(\mathtt{P},2)}$$

The execution of the instrumented semantics in Figure 4 with the initial state $(rhs(\mathtt{MAIN}), \varnothing)$ *produces the computation of Figure 5. Here, for clarity, each computation step is labeled with the applied rule (*EC 4 *means* External Choice 4*); in each state, the second component denotes the set of specification positions already evaluated. Note that the first rule applied is (*Synchronized Parallelism 3*) to the initial expression* $rhs(\mathtt{MAIN})$*. This computation corresponds to the execution of the right branch of the choice (i.e.,* $\mathtt{a} \rightarrow \mathtt{STOP}$*). The final state is* $(\bot \underset{\{\mathtt{a}\}}{\parallel} \bot, \omega_4)$

where $\omega_4 = \{(\texttt{MAIN}, \Lambda), (\texttt{MAIN}, 1.1), (\texttt{MAIN}, 1), (\texttt{MAIN}, 1.2), (\texttt{MAIN}, 2), (\texttt{MAIN}, 2.2.1), (\texttt{MAIN}, 2.2), (\texttt{MAIN}, 2.2.2)\}$.

$$\text{(Synchronized Parallelism 3)}\frac{L \quad R}{\mathit{State}\ 1 \xrightarrow{a} \mathit{State}\ 2}\ \text{where}$$

$$\mathit{State}\ 1 = \ ((\texttt{a} \rightarrow \texttt{STOP}) \underset{\{\texttt{a}\}}{\parallel}_{(\texttt{MAIN},\Lambda)} (\texttt{P}\Box(\texttt{a} \rightarrow \texttt{STOP})), \varnothing)$$

$$L = \text{(Prefixing)}\frac{}{(\texttt{a} \rightarrow \texttt{STOP}, \varnothing) \xrightarrow{\texttt{a}} (\texttt{STOP}, \omega_0)}$$
$$\text{where } \omega_0 = \{(\texttt{MAIN}, 1.1), (\texttt{MAIN}, 1)\}$$

$$R = \text{(EC 4)}\frac{\text{(Prefixing)}\dfrac{}{(\texttt{a} \rightarrow \texttt{STOP}, \varnothing) \xrightarrow{\texttt{a}} (\texttt{STOP}, \{(\texttt{MAIN}, 2.2.1), (\texttt{MAIN}, 2.2)\})}}{((\texttt{P}\Box_{(\texttt{MAIN},2)}(\texttt{a} \rightarrow \texttt{STOP})), \varnothing) \xrightarrow{\texttt{a}} (\texttt{STOP}, \omega_1)}$$
$$\text{where } \omega_1 = \{(\texttt{MAIN}, 2.2.1), (\texttt{MAIN}, 2.2)\} \cup \{(\texttt{MAIN}, 2)\}$$

$$\text{and } \mathit{State}\ 2 = \ ((\texttt{STOP} \underset{\{\texttt{a}\}}{\parallel} \texttt{STOP}), \omega_2) \text{ where } \omega_2 = \omega_0 \cup \omega_1 \cup \{(\texttt{MAIN}, \Lambda)\}$$

$$\text{(Synchronized Parallelism 1)}\frac{\text{(STOP)}\dfrac{}{(\texttt{STOP}, \omega_2) \xrightarrow{\tau} (\bot, \omega_2 \cup \{(\texttt{MAIN}, 1.2)\})}}{\mathit{State}\ 2 \xrightarrow{\tau} \mathit{State}\ 3}$$

$$\text{where } \mathit{State}\ 3 = (\bot \underset{\{\texttt{a}\}}{\parallel} \texttt{STOP}, \omega_3) \text{ and } \omega_3 = \omega'' \cup \{(\texttt{MAIN}, 1.2)\} \cup \{(\texttt{MAIN}, \Lambda)\}$$

$$\text{(Synchronized Parallelism 2)}\frac{\text{(STOP)}\dfrac{}{(\texttt{STOP}, \omega_3) \xrightarrow{\tau} (\bot, \omega_3 \cup \{(\texttt{MAIN}, 2.2.2)\})}}{\mathit{State}\ 3 \xrightarrow{\tau} \mathit{State}\ 4}$$

$$\text{where } \mathit{State}\ 4 = (\bot \underset{\{\texttt{a}\}}{\parallel} \bot, \omega_4) \text{ and } \omega_4 = \omega_3 \cup \{(\texttt{MAIN}, 2.2.2)\} \cup \{(\texttt{MAIN}, \Lambda)\}$$

Figure 5: An example of computation with the semantics in Figure 4

Definition 2. (Rewriting Step, Derivation) Given a state of the semantics s, a *rewriting step* for s[5] is the application of a rule of the semantics: $\dfrac{\Theta}{s \xrightarrow{a \text{ or } \tau \text{ or } \checkmark} s'}$ where Θ is a (possibly empty) set of rewriting steps. We say that the rewriting step is *simple* iff Θ is empty. For the sake of concreteness, we often represent the rewriting step for s as $(s \longrightarrow s')$. Given a state of the semantics s_0, we say that the sequence $s_0 \longrightarrow \ldots \longrightarrow s_{n+1}$, $n \geqslant 0$, is a *derivation* of s_0 iff $\forall\ i, 0 \leqslant i \leqslant n, s_i \longrightarrow s_{i+1}$ is a rewriting step. We say that the derivation is *complete* iff there is no possible rewriting step for s_{n+1}. We say that the derivation has *successfully finished* iff the control of s_{n+1} is $\top$.

We use $s_1 \longrightarrow^* s_n$ to denote a feasible (sub)derivation $s_0 \longrightarrow \ldots \longrightarrow s_n$ that leads from s_1 to s_n; and we define $\mathcal{P}os(s_1 \longrightarrow^* s_n) = \{\mathcal{P}os(c_i) \mid 1 \leqslant i \leqslant$

[5]Note that because s is a state, this definition is valid for both semantics presented so far.

$n\}$ where c_i is the control of state s_i. In the following, we will assume that computations start from a distinguished process `MAIN`.

We also define the following notation for a given CSP specification $\mathcal{S}$: $\mathcal{C}alls(\mathcal{S})$ is the set of specification positions for the process calls appearing in $\mathcal{S}$. $\mathcal{P}roc(\mathcal{S})$ is the set of specification positions in left-hand sides of the processes in $\mathcal{S}$ (i.e., $\mathcal{P}roc(\mathcal{S}) = \{\alpha \in \mathcal{P}os(\mathcal{S}) \mid \alpha = (M, 0)\}$).

In addition, given a set of specification positions A, we define $choices(A)$ as the subset of specification positions of operators that are either an internal choice, an external choice or a conditional choice. For instance, in the specification $\mathcal{S}$ of Example 3 we have $\mathcal{C}alls(\mathcal{S}) = \{(\texttt{MAIN}, \texttt{1.1}), (\texttt{MAIN}, \texttt{1.2}), (\texttt{MAIN}, \texttt{2.1}), (\texttt{MAIN}, \texttt{2.2})\}$ and $\mathcal{P}roc(\mathcal{S}) = \{(\texttt{MAIN}, \texttt{0}), (\texttt{BUS}, \texttt{0}), (\texttt{P1}, \texttt{0}), (\texttt{P2}, \texttt{0})\}$.

3. Context-sensitive Synchronized Control Flow Graph

As usual in static analysis, we need a data structure capable of finitely representing the (often infinite) computations of our specifications. Unfortunately, we cannot use the standard *Control Flow Graph* (CFG) [25], nor the *Interprocedural Control Flow Graph* (ICFG) [8] because they cannot represent multiple threads and, thus, they can only be used with sequential programs. In fact, for CSP specifications, being able to represent multiple threads is a necessary but not a sufficient condition. For instance, the *threaded Control Flow Graph* (tCFG) [12, 13] can represent multiple threads through the use of the so called "*start thread*" and "*end thread*" nodes; but it does not handle synchronization between threads. Callahan and Sublok introduced in [4] the *Synchronized Control Flow Graph* (SCFG), a data structure proposed in the context of imperative programs where an event variable is always in one of two states: clear or posted. The initial value of an event variable is always clear. The value of an event variable can be set to posted with the *POST* statement; and a *WAIT* statement suspends execution of the thread that executes it until the specified event variableś value is set to posted. The SCFG explicitly represents synchronization between threads with a special edge for synchronization flows. In words by Callahan and Sublok [4]:

> "*A synchronized control flow graph is a control flow graph augmented with a set E_s of synchronization edges. $(b_1, b_2) \in E_s$ if the last statement in block b_1 is $POST(ev)$ and the first statement in block b_2 is $WAIT(ev)$ where ev is an event variable.*"

In order to adapt the SCFG to CSP, we extend it with the "*start thread*" and "*end thread*" notation from tCFGs. Therefore, in the following we will work with graphs where nodes N are labeled with positions and "*start*", "*end*" labels (we denote the label of node n with $l(n)$). We also use this notation, "*end* $\backslash$" and "*end* $[\![]\!]$", to denote the end of a hiding respectively a renaming operator. In particular, $\forall n \in N, l(n) \in \mathcal{P}os(\mathcal{S}) \cup \mathcal{S}tart(\mathcal{S})$ where:

$$\mathcal{S}tart(\mathcal{S}) = \{\text{"}start\ \alpha\text{"}, \text{"}end\ \alpha\text{"} \mid \alpha \in \mathcal{P}roc(\mathcal{S})\} \cup \{\text{"}end\ \alpha\text{"} \mid \alpha \in \mathcal{P}os(\mathcal{S}) \wedge\ lit(\alpha) \in \{\backslash, [\![]\!]\}\}$$

For the definition of SCFG, we need to provide a notion of *control flow* between the nodes of a labeled graph.

Definition 3. (Control flow) Given a CSP specification $\mathcal{S}$ and a set of labeled nodes N such that $\forall n \in N, l(n) \in \mathcal{P}os(\mathcal{S}) \cup \mathcal{S}tart(\mathcal{S})$, the *control flow* is a binary relation between the nodes in N. Given two nodes n, $n' \in N$, we say that the *control* of n *can pass* to n' iff:

1. $lit(l(n)) \in \{\sqcap, \Box, \not<\not>, |||, ||\} \ \wedge \ l(n) = (M, w) \ \wedge$
   $l(n') \in \{first((M, w.1)), first((M, w.2))\}$
2. $lit(l(n')) = \rightarrow \ \wedge \ l(n') = (M, w) \ \wedge \ l(n) = (M, w.1)$
3. $lit(l(n')) = \ ; \ \wedge \ l(n') = (M, w) \ \wedge \ l(n) \in last((M, w.1))$
4. $lit(l(n)) \in \{\rightarrow, ;\} \ \wedge \ l(n) = (M, w) \ \wedge \ l(n') = first((M, w.2))$
5. $lit(l(n)) \in \{\backslash, [\![]\!]\} \ \wedge \ l(n) = (M, w) \ \wedge \ l(n') = first((M, w.1))$
6. $l(n') =$ "$end\ (M, w)$" $\wedge \ lit((M, w)) \in \{\backslash, [\![]\!]\} \ \wedge \ l(n) \in last((M, w.1))$

where $first((M, w))$ is defined as follows:

$$first((M, w)) = \begin{cases} (M, w.1) & \text{if } lit((M, w)) = \ \rightarrow \\ first((M, w.1)) & \text{if } lit((M, w)) = \ ; \\ (M, w) & \text{otherwise} \end{cases}$$

and where $last((M, w))$ is the set of possible termination points of (M, w):

$$last((M, w)) =$$

$$\begin{cases} \{(M, w)\} & \text{if } lit((M, w)) = SKIP \\ \varnothing & \text{if } lit((M, w)) = STOP \vee (lit((M, w)) \in \{|||, ||\} \wedge \\ & \qquad (last((M, w.1)) = \varnothing \vee last((M, w.2)) = \varnothing)) \\ last((M, w.1)) & \text{if } lit((M, w)) \in \{\sqcap, \Box, \not<\not>\} \vee (lit((M, w)) \in \{|||, ||\} \wedge \\ \cup \ last((M, w.2)) & \qquad last((M, w.1)) \neq \varnothing \wedge last((M, w.2)) \neq \varnothing) \\ last((M, w.2)) & \text{if } lit((M, w)) \in \{\rightarrow, ;\} \\ \{\text{“}end\ (M, w)\text{”}\} & \text{if } lit((M, w)) \in \{\backslash, [\![]\!]\} \end{cases}$$

Rather than using a declarative definition of SCFG, we provide a constructive definition based on the control flow that allows us to compute the SCFG from a CSP specification.

Definition 4. (Synchronized Control Flow Graph) Given a CSP specification $\mathcal{S}$, we define its *Synchronized Control Flow Graph* as a graph $\mathcal{G} = (N, E_c, E_s)$ where nodes $N = \mathcal{P}os(\mathcal{S}) \cup \mathcal{S}tart(\mathcal{S})$. Edges are divided into two groups, *control-flow arcs* (E_c) and *synchronization edges* (E_s). E_s is a set of edges (denoted by $\leftrightarrow$) representing the possible synchronization of two (event) nodes.[6] E_c is a set of arcs (denoted with $\mapsto$) such that, given two nodes $n, n' \in N$, $n \mapsto n' \in E_c$ iff the control of n can pass to n' or one of the following is true:

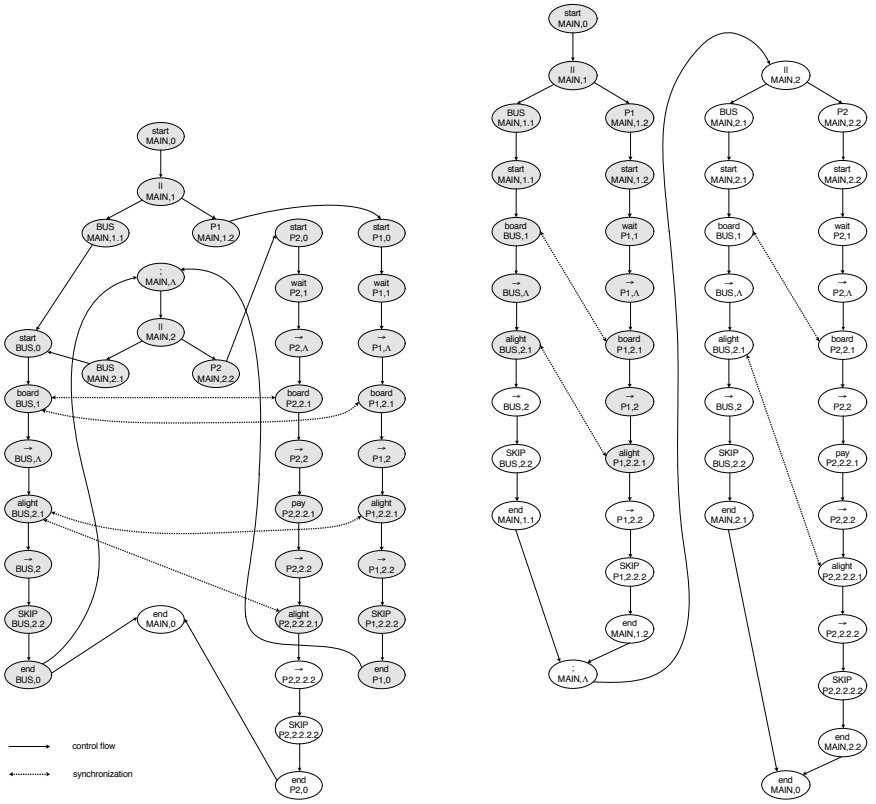
- $lit(l(n)) = M \ \wedge \ l(n') =$ "$start\ (M, 0)$" with $l(n) \in \mathcal{C}alls(\mathcal{S})$
- $l(n) =$ "$start\ (M, 0)$" $\wedge \ l(n') = first((M, \Lambda))$
- $l(n) \in last((M, \Lambda)) \ \wedge \ l(n') =$ "$end\ (M, 0)$"

[6] Computing the events that will synchronize in a specification is a field of research by itself. There are many approaches and algorithms to do this task. In our implementation, we use the technique from [22].

where $last((M, w))$ with $(M, w) \in \mathcal{C}alls(\mathcal{S})$ is defined as $last((M, w)) = \{$"$end\ (P, 0)$"$\}$.

Observe that the size of the SCFG is $\mathcal{O}(n)$ being n the number of positions in the specification. This can be easily proved by showing that there is only one node in the SCFG for each position of the specification, and specification positions are finite and unique. To be fully precise, there is exactly one node for each specification position and two extra nodes for each process (the start process and end process nodes) and one extra node for the hiding and renaming operators (the end hiding and the end renaming). Hence, the size of a SCFG associated to a specification with p processes and n positions with r hiding and renaming operators is $2p + n + r$. The SCFG can be used for slicing CSP specifications as it is described in the following example.

Example 5. *Consider the specification of Example 3 and its associated SCFG shown in Figure 6(a); for the sake of clarity we show the expression represented by each specification position. If we select the node labeled* `(P1,alight)` *and traverse the SCFG backwards in order to identify the nodes on which* `(P1,alight)` *depends, we get the grey nodes of the graph.*



(a) SCFG (b) CSCFG

Figure 6: SCFG and CSCFG of the program in Example 3

The purpose of this example is twofold: on the one hand, it shows that the SCFG can be used for static slicing of CSP specifications. On the other hand,

it shows that it is still too imprecise to be used in practice. The cause of this imprecision is that the SCFG is context-insensitive, because it connects all the calls to the same process with a unique set of nodes. This causes the SCFG to mix different executions of a process with possibly different synchronizations, and, thus it loses precision. For instance, in Example 3 process `BUS` is called twice in different contexts. It is first executed in parallel with `P1` producing the synchronization of their `board` and `alight` events. Then, it is executed in parallel with `P2` producing the synchronization of their `board` and `alight` events. This makes the process `P2` (except nodes $\rightarrow$, `SKIP` and `end P2`) be part of the slice. This is suboptimal because process `P2` is always executed after `P1`.

To the best of our knowledge, there do not exist other data structures that face the problem of representing concurrent and explicitly synchronized computations in a context-sensitive manner. In the rest of this section, we propose a new version of the SCFG, the context-sensitive synchronized control flow graph (CSCFG) which is context-sensitive because it takes into account the different contexts on which a process can be executed.

In contrast to the SCFG, the same specification position can appear multiple times inside a CSCFG. Hence, in the following we will use a refined notion of the $\mathcal{S}tart$ set so that in each "*start* α" and "*end* α" node used to represent a process, α is now any specification position representing a process call instead of a process definition (i.e., not necessarily $\alpha \in \mathcal{P}roc(\mathcal{S})$):

$$\begin{aligned}\mathcal{S}tart(\mathcal{S}) = &\{\text{“}start\ (\texttt{MAIN}, 0)\text{”}, \text{“}end\ (\texttt{MAIN}, 0)\text{”}\}\\ &\cup \{\text{“}start\ \alpha\text{”}, \text{“}end\ \alpha\text{”} \mid \alpha \in \mathcal{C}alls(\mathcal{S})\}\\ &\cup \{\text{“}end\ \alpha\text{”} \mid \alpha \in \mathcal{P}os(\mathcal{S}) \wedge\ lit(\alpha) \in \{\backslash, [\![]\!]\}\}\end{aligned}$$

Using the specification position of the process call allows us to distinguish between different process calls to the same process. Note that we also added to the set the initial and ending nodes of the graph ("*start* $(\texttt{MAIN}, 0)$" and "*end* $(\texttt{MAIN}, 0)$").

Before we properly define the CSCFG, we provide a notion of path and context.

Definition 5. (Path) Given a labeled graph $\mathcal{G} = (N, E_c)$, a *path* between two nodes $n_1, n_k \in N$, represented by $n_1 \mapsto^* n_k$, is a sequence $l(n_1), \ldots, l(n_{k-1})$ such that for all $1 \leqslant i < k$ we have $n_i \mapsto n_{i+1} \in E_c$. The path is *loop-free* if for all $i \neq j$ we have $n_i \neq n_j$.

Definition 6. (Context) Given a labeled graph $\mathcal{G} = (N, E_c)$ and a node $n \in N$, the *context* of n, $\mathcal{C}on(n) = \{m \in N \mid l(m) = \text{“}start\ \alpha\text{”}, \alpha \in \mathcal{C}alls(\mathcal{S})$ and there exists a loop-free path $\pi = m \mapsto^* n$ with "*end* α" $\notin \pi\}$.

Intuitively speaking, the context of a node represents the set of processes in which a particular node is being executed. If we focus on a node n with $l(n) \in \mathcal{C}alls(\mathcal{S})$ we can use the context to identify loops because we have a loop whenever "*start* $l(n)$" $\in \mathcal{C}on(n)$.

The main difference between the SCFG and the CSCFG is that the SCFG represents a process with a single collection of nodes (each specification position in the process is represented with a single node, see Figure 6(a)); in contrast, the CSCFG represents a process with multiple collections of nodes, each collection

representing a different call to this process (i.e., a different context in which it is executed. For instance, see Figure 6(b) where process BUS is represented twice). Therefore, the notion of control flow used in the SCFG is insufficient for the CSCFG, and we need to extend it to also consider the context of process calls.

Definition 7. (Context-sensitive control flow) Given a CSP specification $\mathcal{S}$ and a labeled graph $\mathcal{G} = (N, E_c)$ such that $\forall n \in N, l(n) \in \mathcal{P}os(\mathcal{S}) \cup \mathcal{S}tart(\mathcal{S})$, the *context-sensitive control flow* is a binary relation between the nodes in N. Given two nodes n, $n' \in N$, we say that the *context-sensitive control* of n *can pass* to n', i.e., $n \mapsto n' \in E_c$, iff:

- the *control* of n *can pass* to n', or
- $lit(l(n')) = \ ; \wedge\ l(n') = (M, w)\ \wedge\ l(n) \in last(n'')$ with $n'' \in N\ \wedge\ l(n'') = (M, w.1)$
- $l(n') =$ "$end\ (M, w)$" $\wedge\ lit((M, w)) \in \{\backslash, [\![]\!]\}\ \wedge\ l(n) \in last(n'')$ with $n'' \in N\ \wedge\ l(n'') = (M, w.1)$

where $last(n)$ with $l(n) = (M, w)$ is the set of possible termination points of n:

$$last(n) = \begin{cases} \{(M,w)\} & \text{if } lit((M,w)) = SKIP \\ \varnothing & \text{if } lit((M,w)) = STOP \vee (lit((M,w)) \in \{|||, ||\} \wedge \\ & \quad (last(n1) = \varnothing \vee last(n2) = \varnothing)) \vee \\ & \quad (lit((M,w)) \in \mathcal{C}alls(\mathcal{S}) \wedge \text{"}start\ (M,w)\text{"} \in \mathcal{C}on(n)) \\ last(n1) & \text{if } lit((M,w)) \in \{\sqcap, \Box, \not\leftrightarrow\} \vee (lit((M,w)) \in \{|||, ||\} \wedge \\ \cup\ last(n2) & \quad last(n1) \neq \varnothing \wedge last(n2) \neq \varnothing) \\ last(n2) & \text{if } lit((M,w)) \in \{\rightarrow, ;\} \\ \{\text{"}end\ (M,w)\text{"}\} & \text{if } lit((M,w)) \in \{\backslash, [\![]\!]\} \vee \\ & \quad (lit((M,w)) \in \mathcal{C}alls(\mathcal{S}) \wedge \text{"}start\ (M,w)\text{"} \notin \mathcal{C}on(n)) \end{cases}$$

where $l(n1) = (M, w.1)$ and $l(n2) = (M, w.2)$.

Definition 8. (Context-sensitive Synchronized Control Flow Graph) Given a CSP specification $\mathcal{S}$, we define its *Context-sensitive Synchronized Control Flow Graph* as a graph $\mathcal{G} = (N, E_c, E_l, E_s)$ where nodes N are labeled so that $\forall n \in N, l(n) \in \mathcal{P}os(\mathcal{S}) \cup \mathcal{S}tart(\mathcal{S})$; and $\mathcal{S}tart(\mathcal{S}) = \{$"$start$ (MAIN, 0)", "end (MAIN, 0)"$\}$ $\cup\{$"$start\ \alpha$", "$end\ \alpha$"$|\ \alpha \in \mathcal{C}alls(\mathcal{S})\} \cup \{$"$end\ \alpha$"$|\ lit(\alpha) \in \{\backslash, [\![]\!]\}\}$. Edges are divided into three groups, *control-flow arcs* (E_c), *loop arcs* (E_l) and *synchronization edges* (E_s).

- E_s is a set of edges (denoted by $\leftrightarrow$) representing the possible synchronization of two (event) nodes.[6]
- E_c is a set of arcs (denoted by $\mapsto$) such that, given two nodes $n, n' \in N$, $n \mapsto n' \in E_c$ iff the *context-sensitive control* of n can pass to n' or $l(n) \in \mathcal{C}alls(\mathcal{S}) \wedge l(n') =$ "$start\ l(n)$". And given three nodes $n_1, n_2, n_5 \in N$:
  - $(n_1 \mapsto n_2) \in E_c$ iff $l(n_1) =$ "$start\ \alpha$" $\wedge\ lit(\alpha) = M \wedge l(n_2) = first((M, \Lambda))$,
  - if $l(n_1) \in \mathcal{C}alls(\mathcal{S})$, $l(n_2) =$ "$start\ l(n_1)$" and $n_2 \in \mathcal{C}on(n_1)$ then $\forall n_4 \in N, (n_4 \mapsto n_5)$ with $l(n_4) \in last(n_3)$ with $n_2 \mapsto n_3 \wedge l(n_5) =$ "$end\ \alpha$", and

– $(n_1 \mapsto n_2) \in E_c$ iff $l(n_1) \in last((\texttt{MAIN}, \Lambda)) \wedge l(n_2) =$ "$end\ (\texttt{MAIN}, 0)$".

- E_l is a set of edges (denoted by $\rightsquigarrow$) used to represent loops, i.e., $(n_1 \rightsquigarrow n_2) \in E_l$ iff $l(n_1) \in \mathcal{C}alls(\mathcal{S})$, $l(n_2) =$ "$start\ l(n_1)$" and $n_2 \in \mathcal{C}on(n_1)$.

The CSCFG satisfies the following properties: (i) Two nodes can have the same label. (ii) Every node whose label belongs to $\{$"$start\ \alpha$"$\mid \alpha \in \mathcal{C}alls(\mathcal{S})\}$ has one and only one incoming arc in E_c. (iii) Every process call node has one and only one outgoing arc that belongs to either E_c or E_l.

The key difference between the SCFG and the CSCFG is that the latter unfolds every process call node except those that belong to a loop. This is very convenient for slicing because every process call that is executed in a different context is unfolded and represented with a different subgraph, thus, slicing does not mix computations. Moreover, it allows us to deal with recursion and, at the same time, it prevents infinite unfolding of process calls thanks to the use of loop arcs. Note that loop arcs are only used when the context is repeated (this is ensured by item 3 of the definition). Note also that loops are unfolded only once because the second time they are going to be unfolded the context of the process call node is repeated, and thus a loop arc is used to prevent the unfolding. Properties 2 and 3 ensure finiteness because process calls only have one outgoing arc, and thus, they cannot have a control arc if there is already a loop arc.

The following lemma ensures that the CSCFG is complete: all possible derivations of a CSP specification $\mathcal{S}$ are represented in the CSCFG associated to $\mathcal{S}$.

Lemma 1. *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D} = s_0 \longrightarrow \ldots \longrightarrow s_{n+1}$, $n \geqslant 0$, a* derivation *of $\mathcal{S}$ performed with the instrumented semantics, where $s_0 = (rhs(\texttt{MAIN})_\alpha, \varnothing)$ and $s_{n+1} = (P_\varphi, \omega)$, and $\mathcal{G} = (N, E_c, E_l, E_s)$ the CSCFG associated with $\mathcal{S}$. Then, $\forall \gamma \in \omega : \exists\ \pi = n_1 \mapsto^* n_k \in E_c$, $n_1, n_k \in N$, $k \geqslant 1$, with $l(n_1) = \alpha$ and $l(n_k) = \gamma$.*

This lemma ensures that all derivations are represented in the CSCFG with a path; but, of course, because it is a static representation of any possible execution, the CSCFG also introduces a source of imprecision. This imprecision happens when loop arcs are introduced in a CSCFG, because a loop arc summarizes the rest of a computation with a single collection of nodes, and this collection could mix synchronizations of different iterations. However, note that all process calls of the specification are unfolded and represented with an exclusive collection of nodes, and loop arcs are only introduced if the same call is repeated again. This produces a high level of precision for slicing algorithms.

Because Definition 8 is a declarative definition, it is not very useful for implementors; and hence, we also provide a constructive method that is the basis of our implementation. In particular, the CSCFG can be constructed starting from `MAIN`, and connecting each process call to a subgraph that contains the right-hand side of the called process. Each right-hand side is a new subgraph except if a loop is detected. This process is described in Algorithm 1.

Function $buildGraph(P, Ctx) = (N, E_c, E_l, n_{first}, Last)$ where:

- **((Parameterized) Process call)**. If $P = X_\alpha$ and $\alpha \in \mathcal{C}alls(\mathcal{S})$ then

Algorithm 1 Computing the CSCFG

Input: A specification $\mathcal{S}$ with initial process `MAIN`
Output: The CSCFG $\mathcal{G}$ of $\mathcal{S}$
Begin
Pending=$\{$`MAIN`$\}$
while $Pending \neq \varnothing$ **do**
(1) $(N', E'_c, E_l, n_{first}, Last) = buildGraph(rhs(P), \varnothing)$ where $P \in Pending$
(2) $N = N' \cup \{n_{start}, n_{end}\}$ where n_{start}, n_{end} are fresh,
$l(n_{start})$ = "$start\ (P,0)$" and $l(n_{end})$ = "$end\ (P,0)$"
(3) $E_c = E'_c \cup \{n_{start} \mapsto n_{first}\} \cup \{n_{last} \mapsto n_{end} \mid n_{last} \in Last\}$
(4) $Pending = \{P' \in \mathcal{P}roc(\mathcal{S}) \mid \nexists n \in N : lit(l(n)) = P'\}$
E_s is obtained following the technique from [22]
return $\mathcal{G} = (N, E_c, E_l, E_s)$
End

– if $\exists n_{ctx} \in Ctx$ such that $l(n_{ctx})$ = "$start\ \alpha$" then

$$\begin{array}{l} N = \{n_\alpha\}, \\ E_c = \varnothing, \\ E_l = \{(n_\alpha \rightsquigarrow n_{ctx})\}, \\ n_{first} = n_\alpha \text{ and } Last = \varnothing \end{array}$$

– else

$$\begin{array}{l} N = N_1 \cup \{n_\alpha, n_{start}, n_{end}\}, \\ E_c = E_{c1} \cup E_{c2}, \\ E_l = E_{l1}, \\ n_{first} = n_\alpha \text{ and } Last = \{n_{end}\} \end{array}$$

where

$$\begin{array}{l} n_\alpha, n_{start}, n_{end} \text{ are } fresh \ \wedge\ l(n_\alpha) = \alpha \ \wedge\ l(n_{start}) = \text{"}start\ \alpha\text{"} \\ \wedge\ l(n_{end}) = \text{"}end\ \alpha\text{"} \ \wedge\ X = Q \in \mathcal{S} \ \wedge \\ (N_1, E_{c1}, E_{l1}, n_{first1}, Last_1) = buildGraph(Q, Ctx \cup \{n_{start}\}) \ \wedge \\ E_{c2} = \{(n_\alpha \mapsto n_{start}), (n_{start} \mapsto n_{first1})\} \cup \\ \qquad \{(n_{last} \mapsto n_{end}) \mid n_{last} \in Last_1\}. \end{array}$$

- **(Prefixing)**. If $P = X_\alpha \rightarrow_\beta Q$ and $X \in \{a, a?v, a!v\}$ then

$$\begin{array}{l} N = N_1 \cup \{n_\alpha, n_\beta\}, \\ E_c = E_{c1} \cup \{(n_\alpha \mapsto n_\beta), (n_\beta \mapsto n_{first1})\}, \\ E_l = E_{l1}, \\ n_{first} = n_\alpha \text{ and } Last = Last_1 \end{array}$$

where

$$\begin{array}{l} n_\alpha, n_\beta \text{ are } fresh \ \wedge\ l(n_\alpha) = \alpha \ \wedge\ l(n_\beta) = \beta \ \wedge \\ (N_1, E_{c1}, E_{l1}, n_{first1}, Last_1) = buildGraph(Q, Ctx). \end{array}$$

- **(Choice and parallelism)**. If $P = Q\ X_\alpha\ R$ and $X \in \{\sqcap, \Box, \not<\!\!>, |||, ||\}$ then

$$\begin{array}{l} N = N_1 \cup N_2 \cup \{n_\alpha\}, \\ E_c = E_{c1} \cup E_{c2} \cup \{(n_\alpha \mapsto n_{first1}), (n_\alpha \mapsto n_{first2})\}, \\ E_l = E_{l1} \cup E_{l2}, \\ n_{first} = n_\alpha \text{ and} \end{array}$$

$$Last = \begin{cases} Last_1 \cup Last_2 & \text{if } X \in \{\sqcap, \Box, \lhd\rhd\} \vee \\ & \quad (Last_1 \neq \varnothing \wedge Last_2 \neq \varnothing) \\ \varnothing & \text{if } X \in \{|||, ||\} \wedge \\ & \quad (Last_1 = \varnothing \vee Last_2 = \varnothing) \end{cases}$$

where

$$\begin{array}{l} n_\alpha \text{ is } fresh \quad \wedge \quad l(n_\alpha) = \alpha \quad \wedge \\ (N_1, E_{c1}, E_{l1}, n_{first1}, Last_1) = buildGraph(Q, Ctx) \quad \wedge \\ (N_2, E_{c2}, E_{l2}, n_{first2}, Last_2) = buildGraph(R, Ctx). \end{array}$$

- **(Sequential composition)**. If $P = Q \ ;_\alpha \ R$ then

$$\begin{array}{l} N = N_1 \cup N_2 \cup \{n_\alpha\}, \\ E_c = E_{c1} \cup E_{c2} \cup E_{c3} \cup \{(n_\alpha \mapsto n_{first2})\}, \\ E_l = E_{l1} \cup E_{l2}, \\ n_{first} = n_{first1} \text{ and } Last = Last_2 \end{array}$$

where

$$\begin{array}{l} n_\alpha \text{ is } fresh \quad \wedge \quad l(n_\alpha) = \alpha \quad \wedge \\ (N_1, E_{c1}, E_{l1}, n_{first1}, Last_1) = buildGraph(Q, Ctx) \quad \wedge \\ (N_2, E_{c2}, E_{l2}, n_{first2}, Last_2) = buildGraph(R, Ctx) \quad \wedge \\ E_{c3} = \{(n_{last} \mapsto n_\alpha) \mid n_{last} \in Last_1\}. \end{array}$$

- **(Hiding and renaming)**. If $P = Q \ X_\alpha$ and $X \in \{\backslash, [\![]\!]\}$ then

$$\begin{array}{l} N = N_1 \cup \{n_\alpha, n_{end}\}, \\ E_c = E_{c1} \cup E_{c2} \cup \{(n_\alpha \mapsto n_{first1})\}, \\ E_l = E_{l1}, \\ n_{first} = n_\alpha \text{ and } Last = \{n_{end}\} \end{array}$$

where

$$\begin{array}{l} n_\alpha, n_{end} \text{ are } fresh \quad \wedge \quad l(n_\alpha) = \alpha \quad \wedge \quad l(n_{end}) = \text{“}end\ \alpha\text{”} \quad \wedge \\ (N_1, E_{c1}, E_{l1}, n_{first1}, Last_1) = buildGraph(Q, Ctx) \quad \wedge \\ E_{c2} = \{(n_{last} \mapsto n_{end}) \mid n_{last} \in Last_1\}. \end{array}$$

- **(SKIP and STOP)**. If $P = X_\alpha$ and $X \in \{SKIP, STOP\}$ then

$$\begin{array}{l} N = \{n_\alpha\}, \\ E_c = \varnothing, \\ E_l = \varnothing, \\ n_{first} = n_\alpha \text{ and} \end{array}$$

$$Last = \begin{cases} \{n_\alpha\} & \text{if } X = SKIP \\ \varnothing & \text{if } X = STOP \end{cases}$$

where

$$n_\alpha \text{ is } fresh \quad \wedge \quad l(n_\alpha) = \alpha.$$

The following Lemma ensures that the graph produced by Algorithm 1 is a CSCFG.

Lemma 2. *Let $\mathcal{S}$ be a CSP specification. Then, the execution of Algorithm 1 with $\mathcal{S}$ produces a graph $\mathcal{G}$ that is the CSCFG associated with $\mathcal{S}$ according to Definition 8.*

For slicing purposes, the CSCFG is interesting because we can use the edges to determine if a node must be executed or not before another node, thanks to the following properties:

- if $n \mapsto n' \in E_c$ then n must be executed before n' in all executions.
- if $n \rightsquigarrow n' \in E_l$ then n' must be executed before n in all executions.
- if $n \leftrightarrow n' \in E_s$ then n and n' are executed at the same time in all executions.

While the third property is obvious and it follows from the semantics of synchronized parallelism (concretely, from rule (Synchronized Parallelism 3)), the other two properties require proof. The second property follows trivially from the first property and Definition 8, because loop edges only connect a process call node to a node already repeated in the computation. The first property corresponds to Lemma 3.

Lemma 3. *Let $\mathcal{S}$ be a CSP specification and let $\mathcal{G} = (N, E_c, E_l, E_s)$ be the CSCFG associated with $\mathcal{S}$ according to Definition 8. If $n \mapsto n' \in E_c$ then n must be executed before n' in all executions.*

Thanks to the fact that loops are unfolded only once, the CSCFG ensures that all the specification positions inside the loops are in the graph and can be collected by slicing algorithms. For slicing purposes, this representation also ensures that every possibly executed part of the specification belongs to the CSCFG because only loops (i.e., repeated nodes) are missing.

Example 6. *Consider the specification of Example 3 and its associated CSCFG shown in Figure 6(b). If we select the node labeled* `(P1,alight)` *and traverse the CSCFG backwards in order to identify the nodes on which this node depends, we only get the nodes of the graph colored in gray. This particular slice is optimal and much smaller than the slice obtained when we select the same node* `(P1,alight)` *in the SCFG (see Figure 6(a)).*

The CSCFG provides a different representation for each context in which a process call is made. This can be seen in Figure 6(b) where process `BUS` appears twice to account for the two contexts in which it is called. In particular, in the CSCFG we have a fresh node to represent each different process call, and two nodes point to the same process if and only if they are the same call (they are labeled with the same specification position) and they belong to the same loop. This property ensures that the CSCFG is finite.

Lemma 4. *(Finiteness) Given a specification $\mathcal{S}$, its associated CSCFG is finite.*

Example 7. *The specification in Figure 7 makes clear the difference between the SCFG and the CSCFG. While the SCFG only uses one representation for the process* `P` *(there is only one* `start P`*), the CSCFG uses four different representations because* `P` *could be executed in four different contexts. Note that due to the infinite loops, some parts of the graph are not reachable from* `start MAIN`*; i.e., there is no possible control flow to* `end MAIN`*.*

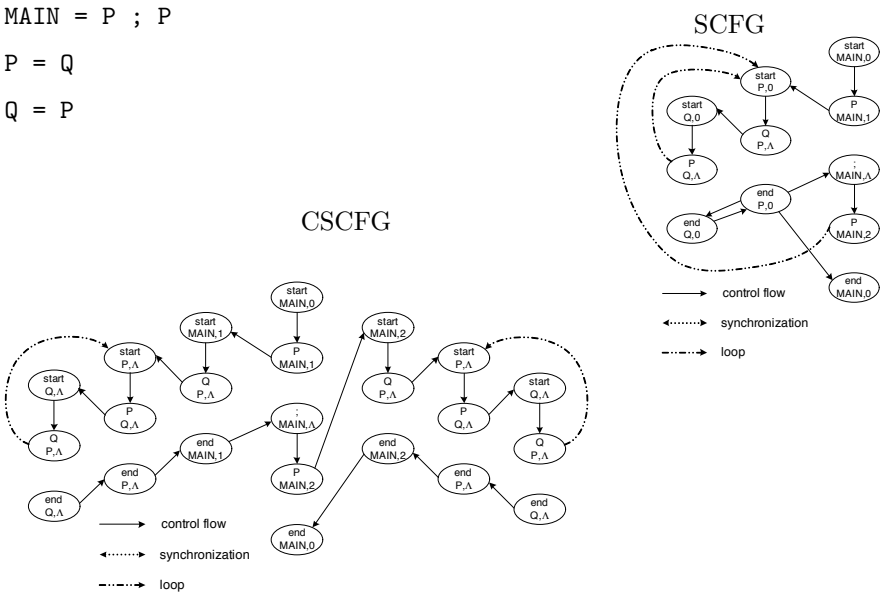


Figure 7: SCFG and CSCFG representing an infinite computation

4. Static Slicing of CSP Specifications

We want to perform two kinds of analysis. Given a point in the specification, we want, on the one hand, to determine what parts of the specification MUST be executed before (MEB) it (in every possible execution); and, on the other hand, we want to determine what parts of the specification COULD be executed before (CEB) it (in any possible execution). Both analyses are closely related but they must be computed differently. While MEB is mainly based on backward slicing, CEB is mainly based on forward slicing to explore what could be executed in parallel processes.

We can now formally define our notion of slicing criterion.

Definition 9. (Slicing Criterion) Given a specification $\mathcal{S}$, a *slicing criterion* is a specification position $\mathcal{C} \in \mathcal{P}os(\mathcal{S})$.

Clearly, the slicing criterion points to a set of nodes in the CSCFG, because the same specification position can happen in different contexts and, thus, it is represented in the CSCFG with different nodes. As an example, consider the slicing criterion `(BUS,alight)` for the specification in Example 3, and observe in its CSCFG in Figure 6(b) that two different nodes are identified by the slicing criterion.

This means that a slicing criterion $\mathcal{C}$ is used to produce a slice with respect to *all* possible executions of $\mathcal{C}$. We use the function $nodes(\mathcal{C})$ to refer to all the nodes in the CSCFG identified by the slicing criterion $\mathcal{C}$. Formally, given a CSCFG $\mathcal{G} = (N, E_c, E_l, E_s)$,

$$nodes(\mathcal{C}) = \{n \in N \mid l(n) = \mathcal{C} \wedge \texttt{MAIN} \mapsto^* n \wedge \nexists\, n' \in N \mid l(n') = \mathcal{C} \text{ and } n' \mapsto^* n\}$$

Note that the slicing criterion could point to nodes that are not reachable from MAIN such as dead code (see, e.g., Figure 7). Therefore, we force $nodes(\mathcal{C})$ to exclude these nodes so that only feasible computations (starting from MAIN) are considered. Moreover, the slicing criterion could also point to different nodes that represent the same specification position that is executed many times in a (sub)computation (see, e.g., specification position $(\mathtt{P}, \Lambda)$ in the CSCFG of Figure 7). Thus, we only select the first occurrence of this specification position in the computation.

Given a slicing criterion (M, w), we use the CSCFG to approximate MEB and CEB. Computing correct slices is known as an undecidable problem even in the sequential setting (see, e.g., [26]). Therefore, our MEB and CEB analyses are an over-approximation. In this section we introduce lemmas to ensure the completeness of the analyses.

Regarding the MEB analysis, one could think that a simple backwards traversal of the graph from $nodes(\mathcal{C})$ would produce a correct slice. Nevertheless, this would produce a rather imprecise slice because this would include both branches of the choices in the path from MAIN to $\mathcal{C}$ even if they do not need to be executed before $\mathcal{C}$ (consider for instance the process ((a→SKIP)□(b→SKIP));P and the slicing criterion P). The union of paths from MAIN to $nodes(\mathcal{C})$ is not a solution, either, because it would be too imprecise by including in the slice parts of code that are executed before the slicing criterion only in some executions. For instance, in the process (b→a→SKIP)□(c→a→SKIP), c belongs to one of the paths to a, but it must be executed before a or not depending on the choice. The intersection of paths is not a solution, either, as it can be seen in the process a→((b→SKIP)||(c→SKIP));P where b must be executed before P, but it does not belong to all the paths from MAIN to P.

Before we introduce an algorithm to compute MEB, we need to formally define the notion of MEB slice.

Definition 10. (MEB Slice) Given a specification $\mathcal{S}$ with an associated CSCFG $\mathcal{G} = (N, E_c, E_l, E_s)$, and a slicing criterion $\mathcal{C}$ for $\mathcal{S}$; the *MEB slice* of $\mathcal{S}$ with respect to $\mathcal{C}$ is a subset $\mathcal{P}'$ of $\mathcal{P}os(\mathcal{S})$ such that $\mathcal{P}' = \bigcap\{\omega \mid (\mathtt{MAIN}, \varnothing) \rightarrow^* (P, \omega) \rightarrow (P', \omega') \wedge \mathcal{C} \notin \omega \wedge \mathcal{C} \in \omega'\}$.

Algorithm 2 can be used to compute the MEB analysis. It basically computes for each node in $nodes(\mathcal{C})$ a set containing the part of the specification that must be executed before it. Then, it returns MEB as the intersection of all these sets. Each set is computed with function *buildMeb*, which is an iterative process that takes a node and performs the following actions:

1. It starts with an initial set of nodes computed in (1) by collecting those nodes that were executed just before the initial node (i.e., they are connected to it or to a node synchronized with it with a control arc).
2. The initial set *Meb* is the backwards traversal of the CSCFG from the initial set following control arcs (2).
3. Those nodes that could not be executed before the initial node are added to a blacklist (3) and (4). The nodes in the blacklist are discarded because they are either a successor of the nodes in the slicing criterion (and thus they are executed always after it), or they are executed in a branch of a choice that cannot lead to the slicing criterion. Note that the blacklist in sentence (4) is computed by iteratively collecting all the nodes that are

Algorithm 2 Computing the MEB set

Input: A CSCFG (N, E_c, E_l, E_s) of a specification $\mathcal{S}$ and a slicing criterion $\mathcal{C}$
Output: A slice of $\mathcal{S}$

Function *buildMeb*(n) :=
(1) $init := \{n' \mid (n' \mapsto o) \in E_c\}$ where $o \in \{n\} \cup \{o' \mid (o' \leftrightarrow n) \in E_s\}$
(2) $Meb := \{o \in (\texttt{MAIN} \mapsto^* m) \mid m \in init\}$
(3) $blacklist := \{n\} \cup \{p \in N \backslash Meb \mid (o \mapsto p) \in E_c$ with $lit(l(o)) \in \{\sqcap, \square, |||\}$, $o \in Meb$ and $\nexists\ q \in Meb$ such that q is reachable from p following control or loop arcs$\}$
repeat
(4) $blacklist := blacklist \cup \{p \in N \mid o \mapsto^* p$ with $o \in blacklist\}$
$\cup \{p \in N \mid (o \leftrightarrow p) \in E_s$ with $o \in blacklist$ and $\nexists (p \leftrightarrow p') \in E_s$ with $p' \notin blacklist\}$
until a fix point is reached
(5) $pending := \{q \in N \backslash (blacklist \cup Meb) \mid (q \leftrightarrow r) \in E_s$ with $r \in Meb$ or $((q \rightsquigarrow r) \in E_l$ and $\forall\ s \in (r \mapsto^* q)\ .\ (\exists (s \leftrightarrow t) \in E_s$ with $t \in Meb$ or $\nexists (s \leftrightarrow t) \in E_s))\}$
(6) while $\exists\ m \in pending$ do
(7) $Meb := Meb \cup \{m\} \cup \{o \in N \backslash Meb \mid (p \mapsto^+ o \mapsto^* m)$ with $p \in Meb\}$
(8) $sync := \{q \in N \backslash (blacklist \cup Meb \cup pending) \mid (q \leftrightarrow r) \in E_s$ with $r \in Meb$ or $((q \rightsquigarrow r) \in E_l$ and $\forall\ s \in (r \mapsto^* q)\ .\ (\exists (s \leftrightarrow t) \in E_s$ with $t \in Meb$ or $\nexists (s \leftrightarrow t) \in E_s))\}$
(9) $pending := (pending \backslash Meb) \cup sync$
(10) return *Meb*

Return: $MEB(\mathcal{S}, \mathcal{C}) = \bigcap_{n \in nodes(\mathcal{C})} \{l(n') \mid n' \in buildMeb(n)\}$

a (control) successor of the nodes in the previous blacklist (initially the slicing criterion); and it also adds to the blacklist those nodes that are only synchronized with nodes in the blacklist.

4. A set of *pending* nodes that should be considered is computed in (5). This set contains nodes that are synchronized with the nodes in *Meb* (thus they are executed at the same time). Therefore, synchronizations are followed in order to reach new nodes that must be executed before the slicing criterion (7) and (8). These steps are repeated until no new nodes are reached. This is controlled with the set *pending* (6) and (9).

The algorithm always terminates as stated in the following lemma.

Theorem 8 (Termination of MEB)**.** *The MEB analysis performed by Algorithm 2 terminates.*

Theorem 9 (Completeness of MEB)**.** *Let $\mathcal{S}$ be a specification, $\mathcal{C}$ a slicing criterion for $\mathcal{S}$, and let $\mathcal{MEB}$ be the MEB slice of $\mathcal{S}$ with respect to $\mathcal{C}$. Then, $\mathcal{MEB} \subseteq MEB(\mathcal{S}, \mathcal{C})$.*

The CEB analysis computes the set of nodes in the CSCFG that could be executed before a given node n. This means that all those nodes that must be executed before n are included, but also those nodes that are executed before

n in some executions, and they are not in other executions (e.g., due to non-synchronized parallelism). Formally,

Definition 11. (CEB Slice) Given a specification $\mathcal{S}$ with an associated CSCFG $\mathcal{G} = (N, E_c, E_l, E_s)$, and a slicing criterion $\mathcal{C}$ for $\mathcal{S}$; the *CEB slice* of $\mathcal{S}$ with respect to $\mathcal{C}$ is a subset $\mathcal{P}'$ of $\mathcal{P}os(\mathcal{S})$ such that $\mathcal{P}' = \bigcup\{\omega \mid (\texttt{MAIN}, \varnothing) \rightarrow^* (P, \omega) \rightarrow (P', \omega') \wedge \mathcal{C} \notin \omega \wedge \mathcal{C} \in \omega'\}$.

Therefore, $MEB(\mathcal{S}, \mathcal{C}) \subseteq CEB(\mathcal{S}, \mathcal{C})$.

The graph $CEB(\mathcal{S}, \mathcal{C})$ can be computed with Algorithm 3 that, roughly, traverses the CSCFG forwards following all the paths that could be executed in parallel to nodes in $MEB(\mathcal{S}, \mathcal{C})$. In particular, the algorithm computes for each node in $nodes(\mathcal{C})$ a set containing the part of the specification that could be executed before it. Then, it returns CEB as the union of all these sets. Each set is computed with function *buildCeb*, which proceeds as follows:

1. In sentence (1), it initializes the set *Ceb* with function *buildMeb* (trivially, all those specification positions that must be executed before a node n, could be executed before it).
2. In sentence (2), it initializes the set *loopnodes*. This set represents the nodes that belong to a loop in the computation executed before the slicing criterion was reached. For instance, in the process `A=(a→A)□(b→SKIP)` the left branch of the choice is a loop that could be executed several times before the slicing criterion, say `b`, was executed. Initially, this set contains the first node in a branch of a choice operator that does not belong to *Ceb* but can reach *Ceb* through a loop arc.
3. The set *loopnodes* is computed in the first loop of the algorithm, sentences (4) to (10) and they are finally added to the slice (i.e., *Ceb*). In particular, sentence (11) checks that the whole loop could be executed before the slicing criterion. If some sentence of the loop could not be executed before (e.g., because it is synchronized with an event that must occur after the slicing criterion), then the loop is discarded and not included in the slice.
4. The second loop of the algorithm, sentences (12) to (18), is used to collect all those nodes that could be executed in parallel to the nodes in the slice (in *Ceb*). In particular, it traverses branches executed in parallel to nodes in *Ceb* until a node that could not be executed before the slicing criterion is found. For instance, consider the process $\texttt{A=(a→b→SKIP)}\,||_{\{\texttt{b}\}}\,\texttt{(c→b→SKIP)}$; and let us assume that the slicing criterion is `c`. Similarly to the first loop of the algorithm, the second loop traverses the left branch of the parallelism operator forwards until an event that could not be executed before the slicing criterion is found (in this example, `b`). Therefore, `a→` would be included in the slice.

The algorithms presented can extract a slice from any specification formed with the syntax of Figure 1. However, note that only two operators have a special treatment in the algorithms: choices (because they introduce alternative computations) and synchronized parallelism constructs (because they introduce synchronization). Other operators such as prefixing, interleaving or sequential composition are only taken into account in the CSCFG construction phase; and they can be treated similarly in the algorithm (i.e., they are traversed forwards or backwards by the algorithm when exploring computations).

Algorithm 3 Computing the CEB set

Input: A CSCFG (N, E_c, E_l, E_s) of a specification $\mathcal{S}$ and a slicing criterion $\mathcal{C}$
Output: A slice of $\mathcal{S}$

Function $buildCeb(n) :=$
(1) $Ceb := buildMeb(n)$
(2) $loopnodes := \{p \mid n_1 \mapsto p \mapsto^* n_2 \rightsquigarrow n_3$
with $n_1 \in choices(Ceb)$, $p, n_2 \notin Ceb$ and $n_3 \in Ceb\}$
(3) $candidates := \varnothing$
repeat
(4) if $\exists (m \mapsto m') \in E_c$ with $m \in loopnodes$ and $m' \notin loopnodes$
(5) then if $\exists (m' \leftrightarrow m'') \in E_s$ with $m'' \in Ceb$ or $\nexists (m' \leftrightarrow m'') \in E_s)$
(6) then $loopnodes := loopnodes \cup \{m'\}$
(7) else $candidates := candidates \cup \{m'\}$
(8) if $\exists (m \leftrightarrow m') \in E_s$ and $m, m' \in candidates$
(9) then $loopnodes := loopnodes \cup \{m, m'\}$
(10) $candidates := candidates \backslash \{m, m'\}$
until a fix point is reached
(11) $Ceb := Ceb \cup \{p \in loopnodes \mid \forall\ o \in loopnodes, p \mapsto^* o \rightsquigarrow q$ with $q \in Ceb\}$
(12) $pending := \{m \in N \backslash (Ceb \cup \{n\}) \mid (m' \mapsto m) \in E_c$ and $m' \in Ceb \backslash choices(Ceb)\}$
repeat
(13) if $\exists\ m \in pending \mid (m \leftrightarrow m') \notin E_s$ or $((m \leftrightarrow m') \in E_s$ and $m' \in Ceb)$
(14) then $Ceb := Ceb \cup \{m\}$
(15) $pending := (pending \backslash \{m\}) \cup \{m'' \mid (m \mapsto m'') \in E_c$ and $m'' \notin Ceb\}$
(16) else if $\exists\ m \in pending$ and $(m \leftrightarrow m') \in E_s$ with $m' \in pending$
(17) then $Ceb := Ceb \cup \{m, m'\}$
(18) $pending := (pending \backslash \{m, m'\}) \cup \{p \mid (o \mapsto p) \in E_c$ and $p \notin Ceb$,
with $o \in \{m, m'\}\}$
until a fix point is reached
(19) return Ceb

Return: $CEB(\mathcal{S}, \mathcal{C}) = \bigcup_{n \in nodes(\mathcal{C})} \{l(n') \mid n' \in buildCeb(n)\}$

Theorem 10 (Termination of CEB)**.** *The CEB analysis performed by Algorithm 3 terminates.*

Theorem 11 (Completeness of CEB)**.** *Let $\mathcal{S}$ be a specification, $\mathcal{C}$ a slicing criterion for $\mathcal{S}$, and let $\mathcal{CEB}$ be the CEB slice of $\mathcal{S}$ with respect to $\mathcal{C}$. Then, $\mathcal{CEB} \subseteq CEB(\mathcal{S}, \mathcal{C})$.*

5. Implementation

We have implemented the MEB and CEB analyses and the algorithms to build the CSCFG for ProB. ProB [15] is an animator for the B-Method which also supports other languages such as CSP [3, 16]. ProB has been implemented in Prolog and it is publicly available at `http://www.stups.uni-duesseldorf.de/ProB`.

Our tool is called SOC (which stands for *Slicing Of CSP*) and it is currently integrated, distributed and maintained for Mac, Linux and Windows since the 1.3 release of ProB. In SOC, the slicing process is completely automatic. Once

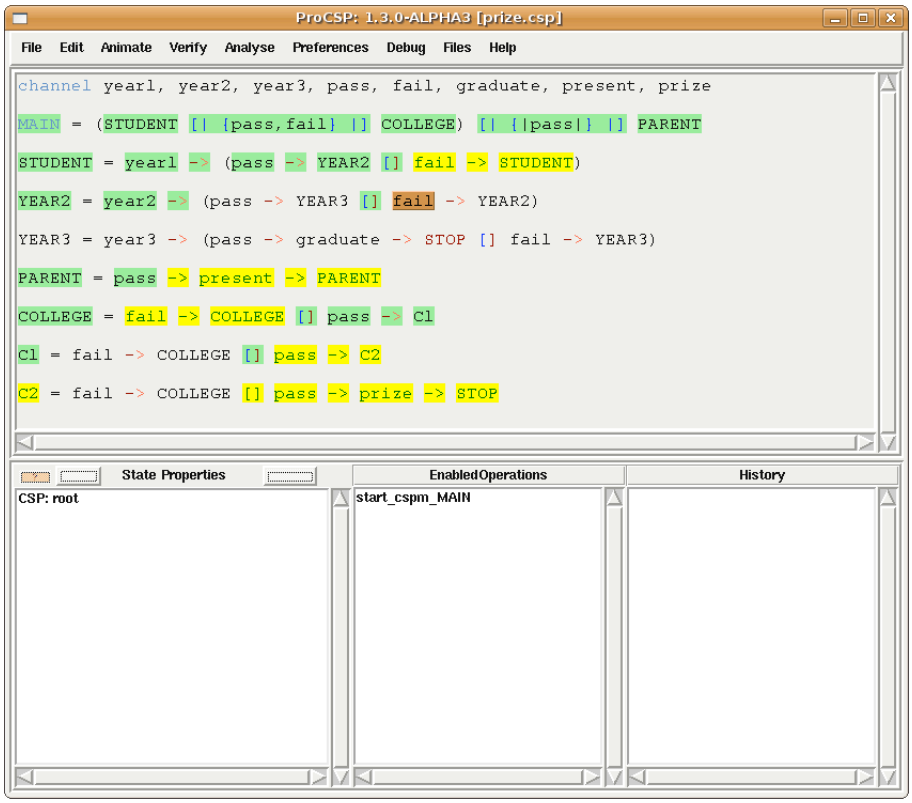


Figure 8: Slice of a CSP specification produced by SOC

the user has loaded a CSP specification, she can select (with the mouse) the event, operator or process call she is interested in. Obviously, this simple action is enough to define a slicing criterion because the tool can automatically determine the process and the source position of interest. Then, the tool internally generates an internal data structure (the CSCFG) that represents all possible computations, and uses the MEB and CEB algorithms to construct the slices. The result is shown to the user by highlighting the part of the specification that must (respectively could) be executed before the specified event. Figure 8 shows a screenshot of the tool showing a slice of the specification in Example 1. SOC also includes a transformation to convert slices into executable programs. This allows us to use SOC for program specialization. The specialized versions produced can be directly executed in ProB.

5.1. Architecture of SOC

SOC has been implemented in Prolog and it has been integrated in ProB. Therefore, SOC can take advantage of ProB's graphical features to show slices to the user. In order to be able to color parts of the code, it has been necessary to implement the source code positions detection in such a way that ProB can color every subexpression that is sliced by SOC.

Figure 9 summarizes the internal architecture of SOC. Note that both the graph compaction module and the slicing module take a CSCFG as input, and

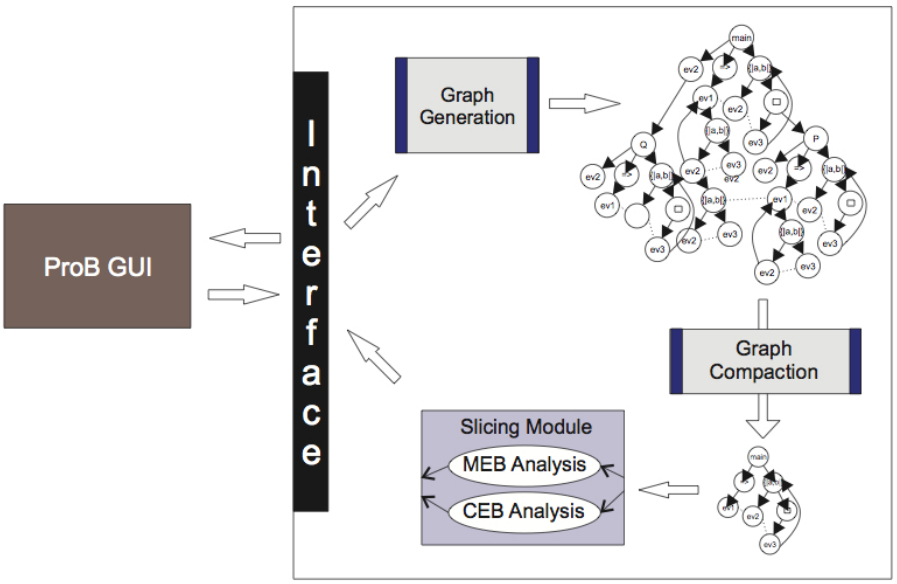


Figure 9: Slicer's Architecture

hence, they are independent of CSP. Apart from the interface module for the communication with ProB, SOC has three main modules that we describe in the following:

Graph Generation

The first task of the slicer is to build a CSCFG. The module that generates the CSCFG from the source program is the only module that is CSP dependent. This means that SOC could be used in other languages by only changing the graph generation module.

Nodes and control and loop arcs are built following Definition 8. For synchronization edges we use an algorithm based on the approach by Naumovich et al. [22]. For efficiency reasons, the implementation of the CSCFG makes some simplifications that reduce the size of the graph. For instance, "*start*" and "*end*" nodes are not present in the graph. Another simplification to reduce the size of the graph is graph compaction (described below).

We have implemented two versions of this module. The first version has the objective of producing a precise analysis. For this purpose, the notion of context described in Definition 6 is used together with the first property of Definition 8. Recall that this property uses the context to introduce loop arcs in the graph whenever a specification position is repeated in a loop. However, this notion of context can produce big CSCFGs with some examples. This implies more memory usage and more time to compute the graphs and the slices. In such cases, the user could be interested in producing the CSCFG as fast as possible; for instance, when the analysis is used as a preprocessing stage of another analysis. Therefore, we have produced a lightweight version to produce a fast analysis when necessary. This second version uses a relaxed notion of context that allows the CSCFG to cut more branches of the graph with loop arcs. The fast analysis replaces property one in Definition 8 by

- There is a special set of *loop arcs* (E_l) denoted with $\rightsquigarrow$. $(n_1 \rightsquigarrow n_2) \in E_l$ iff $l(n_1) \in \mathcal{C}alls(\mathcal{S}) \wedge l(n_2) =$ "$start\ s$" $\wedge\, lit(l(n_1)) = lit(s) \wedge n_2 \in \mathcal{C}on(n_1)$.

which skips the restriction that the specification position of n_1 must be repeated. Therefore, while the *precise* context only introduces a loop arc in the CSCFG when the same specification position is repeated in a branch, the *fast* context introduces a loop arc when the same process call is repeated, even if the specification position of the call is different.

Example 12. *Consider again the CSCFG in Figure 7. This CSCFG corresponds to the precise context, and thus loop arcs are only used when the same specification position is repeated. In contrast, the CSCFG constructed using the fast context uses loop arcs whenever the same process call is repeated (i.e., the literal). It is depicted in Figure 10.*

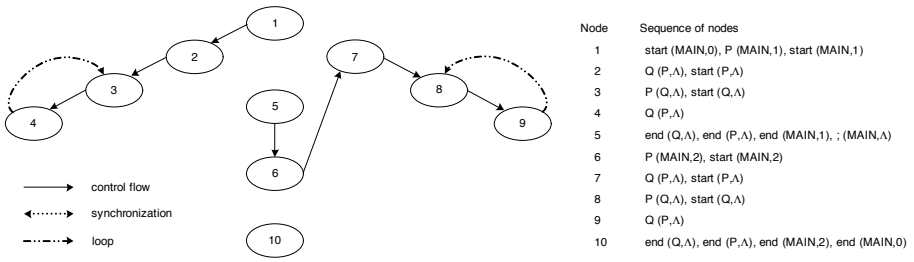


Figure 10: CSCFG of the specification in Figure 7 using the fast context.

Both analyses have been compared with several benchmarks. The results are presented in Section 5.2.

Graph Compaction

For the sake of clarity, the definition of CSCFG proposed does not take into account efficiency. In particular, it includes several nodes that are unnecessary from an implementation point of view. Therefore, we have implemented a module that reduces the size of the CSCFG by removing superfluous nodes and by joining together those nodes that form paths that the slicing algorithms must traverse in all cases. This compaction not only reduces the size of the stored CSCFG, but it also speeds up the slicing process due to the reduced number of nodes to be processed.

For instance, the graph of Figure 11 is the compacted version of the CSCFG in Figure 6(b). Here, e.g., node `2` accounts for the sequence of nodes `BUS` and `start BUS`. The compacted version is a very convenient representation because the reduced data structure speeds up the graph traversal process. In practice, the graph compaction phase reduces the size of the graph up to 40% on average.

Slicing Module

This is the main module of the tool. It is further composed of two submodules that implement the algorithms to perform the MEB and CEB analyses on the compacted CSCFGs. This module extracts two subgraphs from the compacted CSCFG using both MEB and CEB. Then, it extracts from the subgraphs the part of the source code which forms the slice. This information can

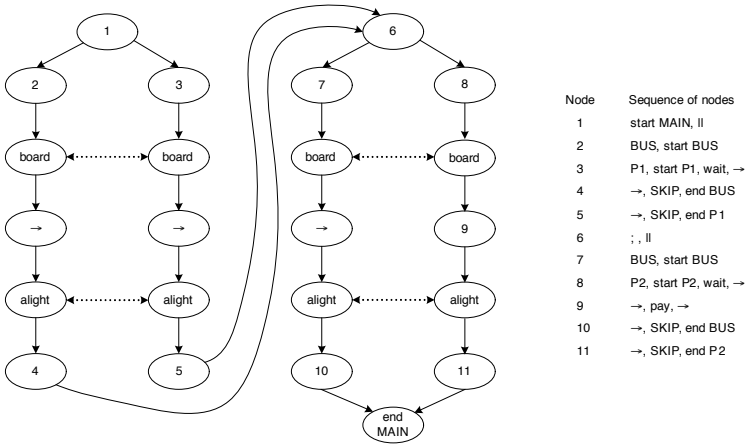


Figure 11: Compacted version of the CSCFG in Figure 6(b)

be extracted directly from the graph because its nodes are labeled with the specification positions to be highlighted. If the user has selected to produce an executable slice, then the slice is further transformed to become executable (it mainly fills gaps in the produced slice in order to respect the syntax of the language). The final result is then returned to ProB in such a way that ProB can either highlight the final slice or save a new CSP executable specification in a file.

5.2. Benchmarking the slicer

In order to measure the performance and the slicing capabilities of our tool, we conducted some experiments over the following benchmarks:

- `ATM.csp`. This specification represents an Automated Teller Machine. The slicing criterion is `(Menu,getmoney)`, i.e., we are interested in determining what parts of the specification must be executed before the menu option `getmoney` is chosen in the ATM.

- `RobotControl.csp`. This example describes a game in which four robots move in a maze. The slicing criterion is `(Referee,winner2)`, i.e., we want to know what parts of the system could be executed before the second robot wins.

- `Buses.csp`. This example describes a bus service with two buses running in parallel. The slicing criterion is `(BUS37, pay90)`, i.e., we are interested in determining what could and could not happen before the user payed at bus 37.

- `Prize.csp`. This is the specification of Example 1. Here, the slicing criterion is `(YEAR2, fail)`, i.e., we are interested in determining what parts of the specification must be executed before the student fails in the second year.

- `Phils.csp`. This is a simple version of the dining philosophers problem. In this example, the slicing criterion is `(PHIL221, DropFork2)`, i.e., we

Table 1: Benchmark time results for the FAST and PRECISE CONTEXT

(a) Benchmark time results for the FAST CONTEXT

Benchmark	CSCFG	MEB	CEB	Total
ATM.csp	805 ms.	36 ms.	67 ms.	908 ms.
RobotControl.csp	277 ms.	39 ms.	21 ms.	337 ms.
Buses.csp	29 ms.	2 ms.	1 ms.	32 ms.
Prize.csp	55 ms.	35 ms.	10 ms.	100 ms.
Phils.csp	72 ms.	12 ms.	4 ms.	88 ms.
TrafficLights.csp	103 ms.	20 ms.	12 ms.	135 ms.
Processors.csp	10 ms.	4 ms.	2 ms.	16 ms.
ComplexSync.csp	212 ms.	264 ms.	38 ms.	514 ms.
Computers.csp	23 ms.	6 ms.	1 ms.	30 ms.
Highways.csp	11452 ms.	100 ms.	30 ms.	11582 ms.

(b) Benchmark time results for the PRECISE CONTEXT

Benchmark	CSCFG	MEB	CEB	Total
ATM.csp	10632 ms.	190 ms.	272 ms.	11094 ms.
RobotControl.csp	2603 ms.	413 ms.	169 ms.	3185 ms.
Buses.csp	25 ms.	1 ms.	0 ms.	26 ms.
Prize.csp	352 ms.	317 ms.	79 ms.	748 ms.
Phils.csp	96 ms.	12 ms.	8 ms.	116 ms.
TrafficLights.csp	2109 ms.	1678 ms.	416 ms.	4203 ms.
Processors.csp	15 ms.	2 ms.	5 ms.	22 ms.
ComplexSync.csp	23912 ms.	552 ms.	174 ms.	24638 ms.
Computers.csp	51 ms.	4 ms.	6 ms.	61 ms.
Highways.csp	58254 ms.	1846 ms.	2086 ms.	62186 ms.

want to know what happened before the second philosopher dropped the second fork.

- `TrafficLights.csp`. This specification defines two cars driving in parallel on different streets with traffic lights for cars controlling. The slicing criterion is `(STREET3,park)`, i.e., we are interested in producing an executable version of the specification in which we could simulate the executions where the second car parks on the third street.

- `Processors.csp`. This example describes a system that, once connected, receives data from two machines. The slicing criterion is `(MACH1,datreq)` to know what parts of the example must be executed before the first machine requests data.

- `ComplexSync.csp`. This specification defines five routers working in parallel. Router i can only send messages to router $i + 1$. Each router can send a broadcast message to all routers. The slicing criterion is `(Process3,keep)`, i.e., we want to know what parts of the system could be executed before router 3 keeps a message.

- `Computers.csp`. This benchmark describes a system in which a user can surf internet and download files. The computer can check wether files are infected by virus. The slicing criterion is `(USER,consult_file)`, i.e., we are interested in determining what parts of the specification must be executed before the user consults a file.

Table 2: Benchmark size results for the FAST and PRECISE CONTEXT

(a) Benchmark size results for the FAST CONTEXT

Benchmark	Ori_CSCFG	Com_CSCFG	(%)	MEB Slice	CEB Slice
ATM.csp	156 nodes	99 nodes	63.46 %	32 nodes	45 nodes
RobotControl.csp	337 nodes	121 nodes	35.91 %	22 nodes	109 nodes
Buses.csp	20 nodes	20 nodes	90.91 %	11 nodes	11 nodes
Prize.csp	70 nodes	52 nodes	74.29 %	25 nodes	42 nodes
Phils.csp	181 nodes	57 nodes	31.49 %	9 nodes	39 nodes
TrafficLights.csp	113 nodes	79 nodes	69.91 %	7 nodes	60 nodes
Processors.csp	30 nodes	15 nodes	50.00 %	8 nodes	9 nodes
ComplexSync.csp	103 nodes	69 nodes	66.99 %	37 nodes	69 nodes
Computers.csp	53 nodes	34 nodes	64.15 %	18 nodes	29 nodes
Highways.csp	103 nodes	62 nodes	60.19 %	41 nodes	48 nodes

(b) Benchmark size results for the PRECISE CONTEXT

Benchmark	Ori_CSCFG	Com_CSCFG	(%)	MEB Slice	CEB Slice
ATM.csp	267 nodes	165 nodes	61.8 %	52 nodes	59 nodes
RobotControl.csp	1139 nodes	393 nodes	34.5 %	58 nodes	369 nodes
Buses.csp	22 nodes	20 nodes	90.91 %	11 nodes	11 nodes
Prize.csp	248 nodes	178 nodes	71.77 %	15 nodes	47 nodes
Phils.csp	251 nodes	56 nodes	22.31 %	9 nodes	39 nodes
TrafficLights.csp	434 nodes	267 nodes	61.52 %	7 nodes	217 nodes
Processors.csp	37 nodes	19 nodes	51.35 %	8 nodes	14 nodes
ComplexSync.csp	196 nodes	131 nodes	66.84 %	18 nodes	96 nodes
Computers.csp	109 nodes	72 nodes	66.06 %	16 nodes	67 nodes
Highways.csp	503 nodes	275 nodes	54.67 %	47 nodes	273 nodes

- `Highways.csp`. This specification describes a net of spanish highways. The slicing criterion is `(HW6,Toledo)`, i.e., we want to determine what cities must be traversed in order to reach Toledo from the starting point.

All the source code and other information about the benchmarks can be found at

`http://www.dsic.upv.es/~jsilva/soc/examples`

For each benchmark, Table 1(a) and Table 1(b) summarize the time spent to generate the compacted CSCFG (this includes the generation plus the compaction phases), to produce the MEB and CEB slices (since CEB analysis uses MEB analysis, CEB's time corresponds only to the time spent after performing the MEB analysis), and the total time. Table 1(a) shows the results when using the fast context and Table 1(b) shows the results associated to the precise context. Clearly, the fast context achieves a significative time reduction. In these tables we can observe that Highways.csp needs more time even though the size of its associated CSCFG is similar to the other examples. Almost all the time needed to construct the CSCFG is used in computing the synchronizations. The high number of synchronizations performed in Highways.csp is the cause of its expensive cost.

Table 2(a) and Table 2(b) summarize the size of all objects participating in the slicing process for both the fast and the precise contexts respectively: Column `Ori_CSCFG` shows the size of the CSCFG of the original program. Observe that the precise context can increase the size of the CSCFG up to four

times with respect to the fast context. Column `Com_CSCFG` shows the size of the compacted CSCFG. Column `(%)` shows the percentage of the compacted CSCFG' size with respect to the original CSCFG. Note that in some examples the reduction is almost 70% of the original size. Finally, columns `MEB Slice` and `CEB Slice` show respectively the size of the MEB and CEB CSCFG' slices. Clearly, CEB slices are always equal or greater than their MEB counterparts.

The CSCFG compaction technique seems to be useful. Experiments show that the size of the original specification is substantially reduced using this technique. The size of both MEB and CEB slices obviously depends on the slicing criterion selected. Table 2(a) and Table 2(b) compare both slices with respect to the same criterion but different contexts and, therefore, they give an idea of the difference between them.

SOC is open and publicly available. All the information related to the experiments, the source code of the benchmarks, the slicing criteria used, the source code of the tool and other material related to the project can be found at

`http://www.dsic.upv.es/~jsilva/soc`

6. Related Work

Program slicing has been already applied to concurrent programs of different programming paradigms, see e.g. [28, 27]. As a result, different graph representations have arisen to represent synchronization. The first proposal of a program slicing method for concurrent programs by Cheng [5] was later improved by Krinke [12, 13] and Nanda [20]. All these approaches are based on the so called *threaded control flow graph* and the *threaded program dependence graph*. Unfortunately, their approaches are not appropriate for slicing CSP, because their work is based on a different kind of synchronization. They use the following concept of *interference* to represent program synchronization.

Definition 12. (Interference) A node $S1$ is *interference* dependent on a node $S2$ if $S2$ defines a variable v, $S1$ uses the variable v and $S1$ and $S2$ execute in parallel.

In CSP, in contrast, a synchronization happens between two processes if the synchronized event is executed at the same time by both processes. In addition, both processes cannot proceed in their executions until they have synchronized. This is the key point that underpin our MEB and CEB analyses. This idea has been already exploited in the *concurrent control flow graph* [7] which allows us to model the phenomenon known as *fully-blocking* semantics where a process sending a message to other process is blocked until the other receives the message and vice versa. This is equivalent to our synchronization model. In these graphs, as in previous approaches (and in conventional program slicing in general), the slicing criterion is a variable in a point of interest, and the slice is formed by the sentences that *influence* this variable due to control and data dependences. For instance, consider the following program fragment:

(1) $read(x)$;
(2) $print(x)$;
(3) $if \quad x > 0$

(4) *then* $y = x - 1$;
(5) *else* $y = 42$;
(6) $print(y)$;
(7) $z = y$;

A slice with respect to $(7, z)$ would contain sentences (1), (3), (4) and (5); because z data depends on y, y data depends on x and (4) and (5) control depend on (3). Sentences (2) and (6) would be discarded because they are print statements and thus, they do not have an influence on z.

In contrast, in our technique, if we select (7) as the slicing criterion, we get sentences (1), (2), (3) and (6) as the MEB slice because these sentences must be executed before the slicing criterion in all executions. The CEB slice would contain the whole program.

Therefore, the purpose of our slicing technique is essentially different from previous work: while other approaches try to answer the question "*what parts of the program can influence the value of this variable at this point?*", our technique tries to answer the question "*what parts of the program must be executed before this point? and what parts of the program can be executed before this point?*". Therefore, our slicing criterion is different, but also the data structure we use for slicing is different. In contrast to previous work, we do not use a PDG like graph, and use instead a CFG like graph, because we focus on control flow rather than control and data dependence.

Despite the problem being undecidable (see Section 1), determining the MEB and CEB slices can be very useful and has many different applications such as debugging, program comprehension, program specialization and program simplification. Surprisingly, to the best of our knowledge, our approach is the first to address the problem in a concurrent and explicitly synchronized context. In fact, the data structure most similar to the CSCFG is the SCFG by Callahan and Sublok [4] (see Section 3 for a detailed description and formalization of this data structure, and a comparison with our CSCFG). Unfortunately, the SCFG does not take the calling context into account and thus it is not appropriate for the MEB and CEB analyses.

Our technique is not the first approach that applies program slicing to CSP specifications. Program slicing has also been applied to CSP by Bruckner and Wehrheim who introduced a method to slice CSP-OZ specifications [2]. Nevertheless, their approach ignores CSP synchronization and focus instead on the OZ's variables. As in previous approaches, their slicing criterion is a LTL formulae constructed with OZ's variables; and they use the standard PDG to compute the slice with a backwards reachability analysis.

7. Conclusions

This work defines two new static analyses that can be applied to languages with explicit synchronization such as CSP. Both techniques are based on program slicing. In particular, we introduce a method to slice CSP specifications, in such a way that, given a CSP specification and a slicing criterion, we produce a slice such that (i) it is a subset of the specification (i.e., it is produced by deleting some parts of the original specification); (ii) it contains all the parts of the specification that must be executed (in any execution) before the slicing

criterion (MEB analysis); and (iii) we can also produce an augmented slice that also contains those parts of the specification that could be executed before the slicing criterion (CEB analysis).

We have presented two algorithms to compute the MEB and CEB analyses based on a new data structure, the CSCFG, that has shown to be more precise than the previously used SCFG. The advantage of the CSCFG is that it cares about contexts, and thus it is able to distinguish between different contexts in which a process is called. This new data structure has been formalized in the paper and compared with the predecessor SCFG.

We have built a tool that implements all the data structures and algorithms defined in the paper; and we have integrated it into the system ProB. This tool is called SOC, and it is now distributed as a part of ProB. Finally, a number of experiments conducted with SOC have been presented and discussed. These experiments demonstrated the usefulness of the technique for different applications such as debugging, program comprehension, program specialization and program simplification.

8. Acknowledgments

We want to thank Mark Fontaine for his help in the implementation. He adapted the ProB module that detects source code positions. We also thank the anonymous referees of LOPSTR'08 [18] and PEPM'09 [17] for many useful comments in a preliminary version of this article.

References

[1] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. In *Computer Networks*, vol. 14, pp. 25–59, 1987.

[2] I. Brückner and H. Wehrheim. Slicing CSP-OZ Specifications for Verification. Technical report, SFB/TR 14 AVACS. Accessible via `http://www.avacs.org/`, 2005.

[3] M. Butler and M. Leuschel. Combining CSP and B for specification and property verification. In *Proceedings of Formal Methods 2005*, LNCS 3582, pp. 221–236, Springer-Verlag, Newcastle, 2005.

[4] D. Callahan and J. Sublok. Static analysis of low-level synchronization. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and Distributed Debugging (PADD'88)*, pp. 100–111, New York, NY, USA, 1988.

[5] J. Cheng. Slicing concurrent programs - a graph-theoretical approach. In *Automated and Algorithmic Debugging*, pp. 223–240, 1993.

[6] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, vol. 9(3), pp. 319–349, 1987.

[7] D. Goswami and R. Mall. Fast Slicing of Concurrent Programs. In *Proceedings of the 6th International Conference on High Performance Computing*, pp. 38–42, 1999.

[8] M. J. Harrold, G. Rothermel, and S. Sinha. Computation of interprocedural control dependence. In *International Symposium on Software Testing and Analysis*, pp. 11–20, 1998.

[9] C. A. R. Hoare. Communicating sequential processes. *Communications ACM*, vol. 26(1), pp. 100–106, 1983.

[10] G. J. Holzmann. Design and Validation of Computer Protocols. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[11] K. M. Kavi, F. T. Sheldon, and B. Shirazi. Reliability analysis of CSP specifications using petri nets and markov processes. In *Proceedings 28th Annual Hawaii International Conference on System Sciences. Software Technology*, vol. 2, pp. 516–524, Wailea, HI, 1995.

[12] J. Krinke. Static slicing of threaded programs. In *Workshop on Program Analysis For Software Tools and Engineering*, pp. 35–42, 1998.

[13] J. Krinke. Context-sensitive slicing of concurrent programs. *ACM SIGSOFT Software Engineering Notes*, vol. 28(5), 2003.

[14] P. Ladkin and B. Simons. Static deadlock analysis for csp-type communications. *Responsive Computer Systems (Chapter 5)*, Kluwer Academic Publishers, 1995.

[15] M. Leuschel and M. Butler. ProB: an automated analysis toolset for the B method. *Journal of Software Tools for Technology Transfer*, vol. 10(2), pp. 185–203, 2008.

[16] M. Leuschel and M. Fontaine. Probing the depths of CSP-M: A new FDR-compliant validation tool. In *Proceedings of the 10th International Conference on Formal Engineering Methods*, LNCS 5256, pp. 278–297. Springer-Verlag, 2008.

[17] M. Leuschel, M. Llorens, J. Oliver, J. Silva, and S. Tamarit. SOC: a slicer for CSP specifications. In *Proceedings of the 2009 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM'09)*, pp. 165–168, Savannah, GA, USA, 2009.

[18] M. Leuschel, M. Llorens, J. Oliver, J. Silva, and S. Tamarit. The MEB and CEB Static Analysis for CSP Specifications. In *Post-proceedings of the 18th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'08), Revised and Selected Papers*, LNCS 5438, pp. 103–118, Springer-Verlag, 2009.

[19] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes. In *Journal of Information and Computation*, Academic Press, vol. 100(1), pp. 1–77, 1992.

[20] M. G. Nanda and S. Ramesh. Slicing concurrent programs. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 180–190, New York, NY, USA, 2000.

[21] V. Natarajan and G. J. Holzmann. Outline for an Operational Semantics of Promela. In *The SPIN Verification Systems. DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, AMS vol. 32, pp. 133152, 1997.

[22] G. Naumovich and G. S. Avrunin. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. *SIGSOFT Software Engineering Notes*, vol. 23(6), pp. 24–34, 1998.

[23] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 2005.

[24] A. W. Roscoe, P. H. B. Gardiner, M. Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood. Hierarchical compression for model-checking CSP or how to check 10^{20} dining philosophers for deadlock. In *Proceedings of the First International Workshop Tools and Algorithms for Construction and Analysis of Systems*, pp. 133–152, 1995.

[25] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, vol. 3, pp. 121–189, 1995.

[26] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, vol. 10(4), pp. 352–357, 1984.

[27] J. Zhao, J. Cheng, and K. Ushijima. Slicing concurrent logic programs. In *Proceedings of the 2nd Fuji International Workshop on Functional and Logic Programming*, pp. 143–162, 1997.

[28] J. Zhao. Slicing aspect-oriented software. In *Proceedings of the 10th IEEE International Workshop on Programming Comprehension*, pp. 251–260, 2002.

A. Proofs of Technical Results

In order to prove Lemmas 1-4 and Theorems 8-11, we first introduce and prove some auxiliary lemmas (Lemmas 5-8) that are needed in their proofs.

Lemma 5. *Let $\mathcal{S}$ be a CSP specification and $s \xrightarrow{\Theta} s'$, a rewriting step performed with the instrumented semantics, with $s = (Ctrl_\alpha, \omega)$ and $s' = (Ctrl'_\varphi, \omega')$. Then, $first(\alpha) \in \omega'$.*

Proof. We proceed by analyzing all possible rules applied in the rewriting step. Considering the semantics in Figure 4 the following cases are possible:

– In the cases of (Process Call), (Parameterized Process Call), (Prefixing), (SKIP), (STOP), (Internal Choice 1 and 2), (Conditional Choice 1 and 2), (External Choice 1, 2, 3 and 4), (Synchronized Parallelism 1, 2 and 3), (Hiding 1, 2 and 3) and (Renaming 1, 2 and 3) the lemma is true straightforwardly from the instrumented semantics definition, Definition 2 (rewriting step) and Definition 3 (control flow).

– In the case of (Synchronized Parallelism 4), this implies that in some previous rewriting steps rules (Synchronized Parallelism 1) and (Synchronized Parallelism 2) were applied. Then, the lemma trivially holds.

– (Sequential Composition 1). If we assume that $Ctrl = P; Q$ only contains one single ; operator, then we have a rewriting step of the form:

$$\frac{(P, \omega) \xrightarrow{a \text{ or } \tau} (P', \omega')}{(P; Q, \omega) \xrightarrow{a \text{ or } \tau} (P'; Q, \omega')}$$

Thus, the lemma holds by applying any of the previous rules to $(P, \omega) \xrightarrow{a \text{ or } \tau} (P', \omega')$. Contrarily, if $Ctrl$ contains more than one ; operator we know that the number of ; operators is finite because $\mathcal{S}$ is finite. Therefore, we can apply rule (Sequential Composition 1) a finite number of times and then any of the previous rules must be applied thus the lemma will eventually hold.

– (Sequential Composition 2). This rule can only be applied after (Sequential Composition 1). Therefore, $first(\alpha) \in \omega'$ because it was included in a previous rewriting step. Hence, the lemma holds.

□

Lemma 6. *Let $\mathcal{S}$ be a CSP specification, $\mathcal{G} = (N, E_c, E_l, E_s)$ the CSCFG associated with $\mathcal{S}$, and $s_i \longrightarrow s_{i+1}$, $0 \leqslant i < n$, a simple rewriting step of $\mathcal{D} = s_0 \longrightarrow \ldots \longrightarrow s_{n+1}$, $n \geqslant 0$, a* derivation *of $\mathcal{S}$ performed with the instrumented semantics, where $s_i = (Ctrl_\alpha, \omega)$ and $s_{i+1} = (Ctrl'_\varphi, \omega')$. Then, $\exists\ \pi = n_j \mapsto^* n_k \in E_c$, $n_j, n_k \in N$, with $l(n_j) = first(\alpha)$ and $l(n_k) = first(\varphi)$.*

Proof. In one simple rewriting step, only one of the following rules can be applied (note that (SKIP), (STOP) and (Synchronized Parallelism 4) cannot be applied because they would always correspond to the last rewriting step $s_n \longrightarrow s_{n+1}$):

– (Process Call) If $Ctrl_\alpha = M_\alpha$, this rule adds to ω the specification position α of M, and the control changes to $rhs(M)_\varphi$. By Definition 8, the context sensitive control of n_j can pass to n_{j+1} with $l(n_j) = first(\alpha) = \alpha \in \mathcal{C}alls(\mathcal{S})$ and $l(n_{j+1}) =$ "*start* $l(n_j)$", i.e., $\alpha \mapsto$ "*start* α"; and the context sensitive control of n_{j+1} can pass to $n_{j+2} = n_k$ with $l(n_{j+2}) = first(\varphi)$, with $\varphi = (lit(\alpha), \Lambda)$, i.e., $first(\alpha) \mapsto$ "*start* α" $\mapsto first(\varphi) \in E_c$.

– (Parameterized Process Call) It is completely analogous to (Process Call).

– (Prefixing) If $Ctrl_\alpha = a_\beta \rightarrow_\alpha P_\varphi$, this rule adds to ω the specification positions of the prefix and the prefixing operator, $\alpha = (M, w)$ and $\beta = \mathit{first}(\alpha) = (M, w.1)$ respectively, and the control changes to P_φ, $\varphi = (M, w.2)$. By Definition 7 (using item 2 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} with $l(n_j) = \beta$ and $l(n_{j+1}) = \alpha$, i.e., $\beta \mapsto \alpha \in E_c$. And by Definition 7 (using item 4 of Definition 3) and Definition 8, the context sensitive control of n_{j+1} can pass to $n_{j+2} = n_k$ with $l(n_{j+1}) = \alpha$ and $l(n_{j+2}) = first(\varphi)$, i.e., $\mathit{first}(\alpha) \mapsto \alpha \mapsto \mathit{first}(\varphi) \in E_c$.

– (Internal Choice 1 and 2) If $Ctrl_\alpha = P \sqcap_\alpha Q$, with this rule the specification position of the choice operator $\alpha = (M, w)$ is added to ω, and one of the two processes P_{φ_1} or Q_{φ_2} is added to the control, with $\varphi_1 = (M, w.1)$ and $\varphi_2 = (M, w.2)$. By Definition 7 (using item 1 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} and to n_{j+2} with $l(n_j) = \mathit{first}(\alpha) = \alpha$ and $l(n_{j+1}) = first(\varphi_1)$ and $l(n_{j+2}) = \mathit{first}(\varphi_2)$, i.e., $\mathit{first}(\alpha) \mapsto \mathit{first}(\varphi_1) \in E_c$ and $\mathit{first}(\alpha) \mapsto \mathit{first}(\varphi_2) \in E_c$.

– (Conditional Choice 1 and 2) These rules are completely analogous to (Internal Choice 1 and 2).

□

For the following lemma we need to provide a notion of height of a rewriting step. The *height* of a rewriting step $s \xrightarrow{\Theta} t$ is defined as:

$$h(s \xrightarrow{\Theta} t) = \begin{cases} 0 & \text{if } \Theta = \varnothing \\ 1 + max(\{h(s' \xrightarrow{\Theta'} t') \mid s' \xrightarrow{\Theta'} t' \in \Theta\}) & \text{otherwise} \end{cases}$$

Lemma 7. *Let $\mathcal{S}$ be a CSP specification, $\mathcal{G} = (N, E_c, E_l, E_s)$ the CSCFG associated with $\mathcal{S}$, and $s_i \xrightarrow{\Theta_i} s_{i+1}$, $i \geqslant 0$, a rewriting step of $\mathcal{D} = s_0 \longrightarrow \ldots \longrightarrow s_{n+1}$, $n \geqslant 0$, a* derivation *of $\mathcal{S}$ performed with the instrumented semantics, where Θ_i is non empty, $s_i = (Ctrl_\alpha, \omega)$ and $s_{i+1} = (Ctrl'_\varphi, \omega')$. Then, $\forall \gamma \in \omega' \backslash \omega : \exists\ \pi = n_j \mapsto^* n_k \in E_c$, $n_j, n_k \in N$, $k \geqslant 1$, with $l(n_j) = \mathit{first}(\alpha)$ and $l(n_k) = \gamma$.*

Proof. Firstly, we know that if Θ_i is not empty, rules (Process Call), (Parameterized Process Call), (Prefixing), (SKIP), (STOP), (Internal Choice 1 and 2), (Conditional Choice 1 and 2) and (Synchronized Parallelism 4) could not be applied. Then, one of the other rules of the instrumented semantics must be applied.

We prove this lemma by induction on the *height* of the rewriting step. The base case happens when the *height* is one, i.e., the rewriting step is of the form $\dfrac{s_j \xrightarrow{a \text{ or } \tau \text{ or } \checkmark} s_{j+1}}{s_i \xrightarrow{a \text{ or } \tau \text{ or } \checkmark} s_{i+1}}$.

In one rewriting step of height one, one of the following rules must be applied:

– (External Choice 1, 2, 3 and 4) If $Ctrl_\alpha = P \Box_\alpha Q$, one of these rules can be applied. If event τ happens, rules (Process Call), (STOP), (Internal Choice 1 or 2) or (Conditional Choice 1 or 2) can be applied. If event a happens, rule (Prefixing) is applied. If event ✓ happens, rules (SKIP) or (Synchronized Parallelism 4) are applied.

  - If (Process Call) is applied, then the rewriting step is:

$$\frac{(P_{\varphi_1}, \omega) \xrightarrow{\tau} (rhs(P)_\beta, \omega \cup \{\varphi_1\})}{(P_{\varphi_1} \Box_\alpha Q_{\varphi_2}, \omega) \xrightarrow{\tau} (rhs(P) \Box_\alpha Q, \omega \cup \{\alpha, \varphi_1\})}$$

By Lemma 6, $first(\varphi_1) \mapsto$ "$start\ \varphi_1$" $\mapsto first(\beta) \in E_c$. By Lemma 5 and by Definition 7 (using item 1 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} and to n_{j+2} with $l(n_j) = first(\alpha) = \alpha$ and $l(n_{j+1}) = first(\varphi_1) = \varphi_1$ and $l(n_{j+2}) = first(\varphi_2)$, i.e., $first(\alpha) \mapsto^* first(\varphi_1) \in E_c$ and $first(\alpha) \mapsto^* first(\varphi_2) \in E_c$.

- If (STOP) is applied, then the rewriting step is:

$$\frac{(STOP_{\varphi_1}, \omega) \xrightarrow{\tau} (\bot, \omega \cup \{\varphi_1\})}{(STOP_{\varphi_1} \Box_\alpha Q_{\varphi_2}, \omega) \xrightarrow{\tau} (\bot \Box_\alpha Q, \omega \cup \{\alpha, \varphi_1\})}$$

  By Lemma 5 and by Definition 7 (using item 1 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} and to n_{j+2} with $l(n_j) = first(\alpha) = \alpha$ and $l(n_{j+1}) = first(\varphi_1) = \varphi_1$ and $l(n_{j+2}) = first(\varphi_2)$, i.e., $first(\alpha) \mapsto^* first(\varphi_1) \in E_c$ and $first(\alpha) \mapsto^* first(\varphi_2) \in E_c$.

- If (Internal Choice 1 or 2) is applied, then the rewriting step is:

$$\frac{(R_{\beta_1} \sqcap_{\varphi_1} S_{\beta_2}, \omega) \xrightarrow{\tau} (R_{\beta_1}, \omega \cup \{\varphi_1\})}{((R \sqcap_{\varphi_1} S) \Box_\alpha Q_{\varphi_2}, \omega) \xrightarrow{\tau} (R \Box_\alpha Q_{\varphi_2}, \omega \cup \{\alpha, \varphi_1\})}$$

  By Lemma 6, $first(\varphi_1) \mapsto first(\beta_1) \in E_c$ and $first(\varphi_1) \mapsto first(\beta_2) \in E_c$. By Lemma 5 and by Definition 7 (using item 1 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} and to n_{j+2} with $l(n_j) = first(\alpha) = \alpha$ and $l(n_{j+1}) = first(\varphi_1) = \varphi_1$ and $l(n_{j+2}) = first(\varphi_2)$, i.e., $first(\alpha) \mapsto^* first(\varphi_1) \in E_c$ and $first(\alpha) \mapsto^* first(\varphi_2) \in E_c$.

- If (Conditional Choice 1 or 2) is applied, then these rules are completely analogous to (Internal Choice 1 and 2).

- If (Prefixing) is applied, then the rewriting step is:

$$\frac{(a_{\alpha_1} \rightarrow_{\varphi_1} R_{\beta_1}, \omega) \xrightarrow{a} (R_{\beta_1}, \omega \cup \{\alpha_1, \varphi_1\})}{((a_{\alpha_1} \rightarrow_{\varphi_1} R_{\beta_1}) \Box_\alpha Q_{\varphi_2}, \omega) \xrightarrow{a} (R_{\beta_1}, \omega \cup \{\alpha, \alpha_1, \varphi_1\})}.$$

  By Lemma 6, $first(\varphi_1) \mapsto \varphi_1 \mapsto first(\beta_1) \in E_c$ where $first(\varphi_1) = \alpha_1$. By Lemma 5 and by Definition 7 (using item 1 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} and to n_{j+2} with $l(n_j) = first(\alpha) = \alpha$ and $l(n_{j+1}) = first(\varphi_1)$ and $l(n_{j+2}) = first(\varphi_2)$, i.e., $first(\alpha) \mapsto^* first(\varphi_1) \in E_c$ and $first(\alpha) \mapsto^* first(\varphi_2) \in E_c$.

- If (SKIP) is applied, then the rewriting step is:

$$\frac{(SKIP_{\varphi_1}, \omega) \xrightarrow{\checkmark} (\top, \omega \cup \{\varphi_1\})}{(SKIP_{\varphi_1} \Box_\alpha Q_{\varphi_2}, \omega) \xrightarrow{\checkmark} (\top, \omega \cup \{\alpha, \varphi_1\})}$$

  By Lemma 5 and by Definition 7 (using item 1 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} and to n_{j+2} with $l(n_j) = first(\alpha) = \alpha$ and $l(n_{j+1}) = first(\varphi_1) = \varphi_1$ and $l(n_{j+2}) = first(\varphi_2)$, i.e., $first(\alpha) \mapsto^* first(\varphi_1) \in E_c$ and $first(\alpha) \mapsto^* first(\varphi_2) \in E_c$.

- If (Synchronized Parallelism 4) is applied, then the rewriting step is:

$$\frac{(\top||\top,\omega) \xrightarrow{\checkmark} (\top,\omega)}{((\top||\top)\Box_\alpha Q,\omega) \xrightarrow{\checkmark} (\top,\omega \cup \{\alpha\})}$$

  If the left(right) process of the choice is $(\top||\top)$, it means that in some previous rewriting steps rules (Synchronized Parallelism 1) and (Synchronized Parallelism 2) were applied. Then, the lemma trivially holds.

– (Synchronized Parallelism 1 and 2) If $Ctrl_\alpha = P \underset{X_\alpha}{\parallel} Q$, this rule can be applied.

  If event $a \notin X$ happens, rule (Prefixing) is applied. If event τ happens, rules (Process Call), (SKIP), (STOP), (Internal Choice 1 or 2), (Conditional Choice 1 or 2) or (Synchronized Parallelism 4) can be applied.

  - If (Prefixing) is applied, then the rewriting step is:

$$\frac{(a_{\alpha_1} \rightarrow_{\varphi_1} P'_{\beta_1},\omega) \xrightarrow{a} (P',\omega \cup \{\alpha_1,\varphi_1\})}{(a_{\alpha_1} \rightarrow_{\varphi_1} P'_{\beta_1} \underset{X_\alpha}{\parallel} Q_{\varphi_2},\omega) \xrightarrow{a} (P' \underset{X}{\parallel} Q,\omega \cup \{\alpha,\alpha_1,\varphi_1\})}\ a \notin X$$

    By Lemma 6, $\mathit{first}(\varphi_1) \mapsto \varphi_1 \mapsto \mathit{first}(\beta_1) \in E_c$ where $\mathit{first}(\varphi_1) = \alpha_1$. By Lemma 5 and by Definition 7 (using item 1 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} and to n_{j+2} with $l(n_j) = \mathit{first}(\alpha) = \alpha$ and $l(n_{j+1}) = first(\varphi_1)$ and $l(n_{j+2}) = \mathit{first}(\varphi_2)$, i.e., $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\varphi_1) \in E_c$ and $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\varphi_2) \in E_c$.

  - If (Process Call) is applied, then the rewriting step is:

$$\frac{(P_{\varphi_1},\omega) \xrightarrow{\tau} (rhs(P)_\beta,\omega \cup \{\varphi_1\})}{(P_{\varphi_1} \underset{X_\alpha}{\parallel} Q_{\varphi_2},\omega) \xrightarrow{\tau} (rhs(P) \underset{X}{\parallel} Q,\omega \cup \{\alpha,\varphi_1\})}$$

    By Lemma 6, $\mathit{first}(\varphi_1) \mapsto$ "$\mathit{start}\ \varphi_1$" $\mapsto \mathit{first}(\beta) \in E_c$. By Lemma 5 and by Definition 7 (using item 1 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} and to n_{j+2} with $l(n_j) = \mathit{first}(\alpha) = \alpha$ and $l(n_{j+1}) = first(\varphi_1) = \varphi_1$ and $l(n_{j+2}) = \mathit{first}(\varphi_2)$, i.e., $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\varphi_1) \in E_c$ and $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\varphi_2) \in E_c$.

  - If (SKIP) is applied, then the rewriting step is:

$$\frac{(SKIP_{\varphi_1},\omega) \xrightarrow{\checkmark} (\top,\omega \cup \{\varphi_1\})}{(SKIP_{\varphi_1} \underset{X_\alpha}{\parallel} Q_{\varphi_2},\omega) \xrightarrow{\tau} (\top \underset{X}{\parallel} Q,\omega \cup \{\alpha,\varphi_1\})}$$

    By Lemma 5 and by Definition 7 (using item 1 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} and to n_{j+2} with $l(n_j) = \mathit{first}(\alpha) = \alpha$ and $l(n_{j+1}) = first(\varphi_1) = \varphi_1$ and $l(n_{j+2}) = \mathit{first}(\varphi_2)$, i.e., $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\varphi_1) \in E_c$ and $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\varphi_2) \in E_c$.

  - If (STOP) is applied, then the rewriting step is:

$$\frac{(STOP_{\varphi_1}, \omega) \xrightarrow{\tau} (\bot, \omega \cup \{\varphi_1\})}{(STOP_{\varphi_1} \underset{X_\alpha}{\parallel} Q_{\varphi_2}, \omega) \xrightarrow{\tau} (\bot \underset{X}{\parallel} Q, \omega \cup \{\alpha, \varphi_1\})}$$

By Lemma 5 and by Definition 7 (using item 1 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} and to n_{j+2} with $l(n_j) = \mathit{first}(\alpha) = \alpha$ and $l(n_{j+1}) = first(\varphi_1) = \varphi_1$ and $l(n_{j+2}) = \mathit{first}(\varphi_2)$, i.e., $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\varphi_1) \in E_c$ and $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\varphi_2) \in E_c$.

- If (Internal Choice 1 or 2) is applied, then the rewriting step is:

$$\frac{(R_{\beta_1} \sqcap_{\varphi_1} S_{\beta_2}, \omega) \xrightarrow{\tau} (R_{\beta_1}, \omega \cup \{\varphi_1\})}{((R \sqcap_{\varphi_1} S) \underset{X_\alpha}{\parallel} Q_{\varphi_2}, \omega) \xrightarrow{\tau} (R \underset{X_\alpha}{\parallel} Q_{\varphi_2}, \omega \cup \{\alpha, \varphi_1\})}$$

By Lemma 6, $\mathit{first}(\varphi_1) \mapsto \mathit{first}(\beta_1) \in E_c$ and $\mathit{first}(\varphi_1) \mapsto \mathit{first}(\beta_2) \in E_c$. By Lemma 5 and by Definition 7 (using item 1 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} and to n_{j+2} with $l(n_j) = \mathit{first}(\alpha) = \alpha$ and $l(n_{j+1}) = first(\varphi_1) = \varphi_1$ and $l(n_{j+2}) = \mathit{first}(\varphi_2)$, i.e., $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\varphi_1) \in E_c$ and $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\varphi_2) \in E_c$.

- If (Conditional Choice 1 or 2) is applied, then these rules are completely analogous to (Internal Choice 1 and 2).
- If (Synchronized Parallelism 4) is applied, then the rewriting step is:

$$\frac{(\top || \top, \omega) \xrightarrow{\checkmark} (\top, \omega)}{((\top || \top) \underset{X_\alpha}{\parallel} Q, \omega) \xrightarrow{\tau} (\top \underset{X}{\parallel} Q, \omega \cup \{\alpha\})}$$

If the left(right) process of the parallelism is $(\top||\top)$, it means that in some previous rewriting steps rules (SP1) and (SP2) were applied. Then, the lemma trivially holds.

– (Synchronized Parallelism 3) If $Ctrl_\alpha = P \underset{X_\alpha}{\parallel} Q$, this rule can be applied. When event $a \in X$ happens, only rule (Prefixing) can be applied. Then the rewriting step is:

$$\frac{(a_{\alpha_1} \rightarrow_{\varphi_1} P'_{\beta_1}, \omega) \xrightarrow{a} (P'_{\beta_1}, \omega \cup \{\alpha_1, \varphi_1\}) \quad (a_{\alpha_2} \rightarrow_{\varphi_2} Q'_{\beta_2}, \omega) \xrightarrow{a} (Q'_{\beta_2}, \omega \cup \{\alpha_2, \varphi_2\})}{((a_{\alpha_1} \rightarrow_{\varphi_1} P'_{\beta_1}) \underset{X_\alpha}{\parallel} (a_{\alpha_2} \rightarrow_{\varphi_2} Q'_{\beta_2}), \omega) \xrightarrow{a} (P' \underset{X}{\parallel} Q', \omega \cup \{\alpha, \alpha_1, \varphi_1, \alpha_2, \varphi_2\})}$$

By Lemma 6, $\mathit{first}(\varphi_1) \mapsto \varphi_1 \mapsto \mathit{first}(\beta_1) \in E_c$ where $\mathit{first}(\varphi_1) = \alpha_1$ and $\mathit{first}(\varphi_2) \mapsto \varphi_2 \mapsto \mathit{first}(\beta_2) \in E_c$ where $\mathit{first}(\varphi_2) = \alpha_2$. By Lemma 5 and by Definition 7 (using item 1 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} and to n_{j+2} with $l(n_j) = \mathit{first}(\alpha) = \alpha$ and $l(n_{j+1}) = first(\varphi_1)$ and $l(n_{j+2}) = \mathit{first}(\varphi_2)$, i.e., $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\varphi_1) \in E_c$ and $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\varphi_2) \in E_c$.

– (Sequential Composition 1) If $Ctrl_\alpha = P;_\alpha Q$, this rule can be applied. When event a happens, only rule (Prefixing) can be applied. If event τ happens, rules (Process Call), (STOP), (Internal Choice 1 or 2) or (Conditional Choice 1 or 2) can be applied.

- If (Prefixing) is applied, then the rewriting step is:

$$\frac{(a_{\alpha_1} \rightarrow_{\varphi_1} P'_{\beta_1}, \omega) \xrightarrow{a} (P'_{\beta_1}, \omega \cup \{\alpha_1, \varphi_1\})}{((a_{\alpha_1} \rightarrow_{\varphi_1} P'_{\beta_1});_\alpha Q_{\varphi_2}, \omega) \xrightarrow{a} (P'; Q, \omega \cup \{\alpha_1, \varphi_1\})}$$

  By Lemma 6, $\mathit{first}(\varphi_1) \mapsto \varphi_1 \mapsto \mathit{first}(\beta_1) \in E_c$ where $\mathit{first}(\varphi_1) = \alpha_1$. By Definition 7 (using item 3 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} with $l(n_j) = \mathit{last}(\varphi_1) = \mathit{last}(\beta_1)$ and $l(n_{j+1}) = \alpha$, i.e., $\mathit{last}(\beta_1) \mapsto \alpha \in E_c$.

- If (Process Call) is applied, then the rewriting step is:

$$\frac{(P_{\varphi_1}, \omega) \xrightarrow{\tau} (rhs(P)_{\beta_1}, \omega \cup \{\varphi_1\})}{(P_{\varphi_1};_\alpha Q_{\varphi_2}, \omega) \xrightarrow{\tau} (rhs(P); Q, \omega \cup \{\varphi_1\})}$$

  By Lemma 6, $\mathit{first}(\varphi_1) \mapsto$ "$\mathit{start}\ \varphi_1$" $\mapsto \mathit{first}(\beta_1) \in E_c$. By Definition 7 (using item 3 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} with $l(n_j) = \mathit{last}(\varphi_1) = \mathit{last}(\beta_1)$ and $l(n_{j+1}) = \alpha$, i.e., $\mathit{last}(\beta_1) \mapsto \alpha \in E_c$.

- If (STOP) is applied, then the rewriting step is:

$$\frac{(STOP_{\varphi_1}, \omega) \xrightarrow{\tau} (\bot, \omega \cup \{\varphi_1\})}{(STOP_{\varphi_1};_\alpha Q_{\varphi_2}, \omega) \xrightarrow{\tau} (\bot; Q, \omega \cup \{\varphi_1\})}$$

  By Definition 7, Definition 8 and Lemma 5, the context sensitive control of n_j can pass to n_{j+1} with $l(n_j) = first(\alpha)$ and $l(n_{j+1}) = \varphi_1$, i.e., $first(\alpha) \mapsto \varphi_1 \in E_c$.

- If (Internal Choice 1 or 2) is applied, then the rewriting step is:

$$\frac{(R_{\beta_1} \sqcap_{\varphi_1} S_{\beta_2}, \omega) \xrightarrow{\tau} (R_{\beta_1}, \omega \cup \{\varphi_1\})}{((R_{\beta_1} \sqcap_{\varphi_1} S_{\beta_2});_\alpha Q_{\varphi_2}, \omega) \xrightarrow{\tau} (R; Q, \omega \cup \{\varphi_1\})}$$

  By Lemma 6, $\mathit{first}(\varphi_1) \mapsto \mathit{first}(\beta_1) \in E_c$ and $\mathit{first}(\varphi_1) \mapsto \mathit{first}(\beta_2) \in E_c$. By Definition 7 (using item 3 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} and to n_{j+2} with $l(n_j) = \mathit{last}(\varphi_1) = \mathit{last}(\beta_1)$, $l(n_{j+1}) = \alpha$ and $l(n_{j+2}) = \mathit{last}(\varphi_2) = \mathit{last}(\beta_2)$, i.e., $\mathit{last}(\beta_1) \mapsto \alpha \in E_c$ and $\mathit{last}(\beta_1) \mapsto \alpha \in E_c$.

- If (Conditional Choice 1 or 2) is applied, then these rules are completely analogous to (Internal Choice 1 and 2).

– (Sequential Composition 2) If $Ctrl_\alpha = P;_\alpha Q$ and this rule can be applied, P will be $SKIP$ or $\top||\top$, i.e., rules (SKIP) or (Synchronized Parallelism 4) can be applied.

  - If (SKIP) is applied, then the rewriting step is:

$$\frac{(SKIP_{\varphi_1}, \omega) \xrightarrow{\checkmark} (\top, \omega \cup \{\varphi_1\})}{(SKIP_{\varphi_1};_\alpha Q_{\varphi_2}, \omega) \xrightarrow{\tau} (Q, \omega \cup \{\alpha, \varphi_1\})}$$

    By Definition 7 (using item 4 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} with $l(n_j) = \alpha$ and $l(n_{j+1}) = \mathit{first}(\varphi_2)$, i.e., $\alpha \mapsto \mathit{first}(\varphi_2) \in E_c$.

  - If (Synchronized Parallelism 4) is applied, then the rewriting step is:

$$\frac{(\top||\top,\omega) \xrightarrow{\checkmark} (\top,\omega)}{((\top||\top);_\alpha Q_{\varphi_2},\omega) \xrightarrow{\tau} (Q,\omega \cup \{\alpha\})}$$

By Definition 7 (using item 4 of Definition 3) and Definition 8, the context sensitive control of of n_j can pass to n_{j+1} with $l(n_j) = \alpha$ and $l(n_{j+1}) = \mathit{first}(\varphi_2)$, i.e., $\alpha \mapsto \mathit{first}(\varphi_2) \in E_c$.

– (Hiding 1) If $Ctrl_\alpha = P\backslash_\alpha B$, this rule can be applied. When event $a \in B$ happens, only rule (Prefixing) can be applied. The rewriting step is:

$$\frac{(a_{\alpha_1} \rightarrow_{\varphi_1} P'_{\beta_1},\omega) \xrightarrow{a} (P'_{\beta_1},\omega \cup \{\alpha_1,\varphi_1\})}{((a_{\alpha_1} \rightarrow_{\varphi_1} P'_{\beta_1})\backslash_\alpha B,\omega) \xrightarrow{\tau} (P'\backslash_\alpha B,\omega \cup \{\alpha,\alpha_1,\varphi_1\})}$$

By Lemma 6, $\mathit{first}(\varphi_1) \mapsto \varphi_1 \mapsto \mathit{first}(\beta_1) \in E_c$ where $\mathit{first}(\varphi_1) = \alpha_1$. By Definition 7 (using item 5 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} with $l(n_j) = first(\alpha) = \alpha$ and $l(n_{j+1}) = first(\varphi_1)$, i.e., $\alpha \mapsto^* \mathit{first}(\varphi_1) \in E_c$.

– (Hiding 2) If $Ctrl_\alpha = P\backslash_\alpha B$, this rule can be applied. When event $a \notin B$ happens, only rule (Prefixing) can be applied. If event τ happens, rules (Process Call), (STOP), (Internal Choice 1 or 2) or (Conditional Choice 1 or 2) can be applied.

- If (Prefixing) is applied, then the rewriting step is:

$$\frac{(a_{\alpha_1} \rightarrow_{\varphi_1} P'_{\beta_1},\omega) \xrightarrow{a} (P'_{\beta_1},\omega \cup \{\alpha_1,\varphi_1\})}{((a_{\alpha_1} \rightarrow_{\varphi_1} P'_{\beta_1})\backslash_\alpha B,\omega) \xrightarrow{a} (P'\backslash_\alpha B,\omega \cup \{\alpha,\alpha_1,\varphi_1\})}\ a \notin B$$

  By Lemma 6, $\mathit{first}(\varphi_1) \mapsto \varphi_1 \mapsto \mathit{first}(\beta_1) \in E_c$ where $\mathit{first}(\varphi_1) = \alpha_1$. By Definition 7 (using item 5 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} with $l(n_j) = first(\alpha) = \alpha$ and $l(n_{j+1}) = first(\varphi_1)$, i.e., $\alpha \mapsto^* \mathit{first}(\varphi_1) \in E_c$.

- If (STOP) is applied, then the rewriting step is:

$$\frac{(STOP_{\varphi_1},\omega) \xrightarrow{\tau} (\bot,\omega \cup \{\varphi_1\})}{(STOP_{\varphi_1}\backslash_\alpha B,\omega) \xrightarrow{\tau} (\bot\backslash_\alpha B,\omega \cup \{\alpha,\varphi_1\})}$$

  By Definition 7 (using item 5 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} with $l(n_j) = \alpha$ and $l(n_{j+1}) = first(\varphi_1) = \varphi_1$, i.e., $\alpha \mapsto^* \mathit{first}(\varphi_1) \in E_c$.

- If (Internal Choice 1 or 2) is applied, then the rewriting step is:

$$\frac{(R_{\beta_1} \sqcap_{\varphi_1} S_{\beta_2},\omega) \xrightarrow{\tau} (R_{\beta_1},\omega \cup \{\varphi_1\})}{((R_{\beta_1} \sqcap_{\varphi_1} S_{\beta_2})\backslash_\alpha B,\omega) \xrightarrow{\tau} (R_{\beta_1}\backslash_\alpha B,\omega \cup \{\alpha,\varphi_1\})}$$

  By Lemma 6, $\mathit{first}(\varphi_1) \mapsto \mathit{first}(\beta_1) \in E_c$ and $\mathit{first}(\varphi_1) \mapsto \mathit{first}(\beta_2) \in E_c$. By Definition 7 (using item 5 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} with $l(n_j) = first(\alpha) = \alpha$ and $l(n_{j+1}) = first(\varphi_1) = \varphi_1$, i.e., $\alpha \mapsto^* \mathit{first}(\varphi_1) \in E_c$.

- If (Conditional Choice 1 or 2) is applied, then these rules are completely analogous to (Internal Choice 1 and 2).

– (Hiding 3) If $Ctrl_\alpha = P\backslash_\alpha B$ and this rule can be applied, P will be $SKIP$ or $\top||\top$, i.e., rules (SKIP) or (Synchronized Parallelism 4) can be applied.

- If (SKIP) is applied, then the rewriting step is:

$$\frac{(SKIP_{\varphi_1}, \omega) \xrightarrow{\checkmark} (\top, \omega \cup \{\varphi_1\})}{(SKIP_{\varphi_1}\backslash_\alpha B, \omega) \xrightarrow{\checkmark} (\top, \omega \cup \{\alpha, \varphi_1\})}$$

  By Definition 7 (using item 5 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} with $l(n_j) = first(\alpha) = \alpha$ and $l(n_{j+1}) = first(\varphi_1) = \varphi_1$, i.e., $\alpha \mapsto^* first(\varphi_1) \in E_c$.

- If (Synchronized Parallelism 4) is applied, then the rewriting step is:

$$\frac{(\top \|_{\varphi_1} \top, \omega) \xrightarrow{\checkmark} (\top, \omega)}{((\top \|_{\varphi_1} \top)\backslash_\alpha B, \omega) \xrightarrow{\checkmark} (\top, \omega \cup \{\alpha\})}$$

  If the process $(\top||\top)$ is in the control, it means that in some previous rewriting step rules (Synchronized Parallelism 1), (Synchronized Parallelism 2) and/or (Synchronized Parallelism 3) were applied and $\varphi_1 \in \omega$. By Definition 7 (using item 5 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_{j+1} with $l(n_j) = first(\alpha) = \alpha$ and $l(n_{j+1}) = first(\varphi_1) = \varphi_1$, i.e., $\alpha \mapsto^* first(\varphi_1) \in E_c$.

– (Renaming 1, 2 and 3) These rules are completely analogous to (Hiding 1, 2 and 3).

We assume as the induction hypothesis that the lemma holds in rewriting steps of height n, and we prove that it also holds in a rewriting step of height $n + 1$, i.e., the rewriting step is of the form:

$$\frac{\Theta}{\dfrac{s_j \xrightarrow{a \text{ or } \tau \text{ or } \checkmark} s_{j+1}}{s_i \xrightarrow{a \text{ or } \tau \text{ or } \checkmark} s_{i+1}}}$$

where Θ is of height $n - 1$ ($n \geqslant 2$), $s_i = (Ctrl_\alpha, \omega)$, $s_j = (Ctrl_\beta, \omega)$, $s_{j+1} = (Ctrl'_\delta, \omega')$ and $s_{i+1} = (Ctrl''_\varphi, \omega'')$.

In order to prove this lemma, we take advantage of the induction hypothesis that ensures that $\forall \gamma \in \omega'\backslash\omega : \exists\ \pi = n_j \mapsto^* n_k \in E_c$, $n_j, n_k \in N$, $k \geqslant 1$, with $l(n_j) = first(\beta)$ and $l(n_k) = \gamma$. Therefore, we need to prove that:

1. $\exists\ \pi_1 = n_i \mapsto^* n_j \in E_c$, $n_i \in N$, with $l(n_i) = first(\alpha)$, and
2. $\forall \gamma \in \omega''\backslash\omega' : \exists\ \pi_2 = n_i \mapsto^* n_k \in E_c$.

In order to prove, item 1, one of the following rules can be applied:

– (External Choice 1, 2, 3 and 4) Trivially, using item 1 of Definition 3, $first(\alpha) \mapsto^* first(\beta) \in E_c$.

– (Synchronized Parallelism 1, 2 and 3) These rules are completely analogous to (External Choice 1, 2, 3 and 4).

– (Hiding 1 and 2) Trivially, using item 5 of Definition 3, $\mathit{first}(\alpha) \mapsto^* \mathit{first}(\beta) \in E_c$.

– (Renaming 1 and 2) These rules are completely analogous to (Hiding 1 and 2).

– (Sequential Composition 1 and 2) In these cases $first(\alpha) = first(\beta)$ and the lemma trivially holds.

In order to prove, item 2, we consider the following two cases:

– One of these rules is applied: (External Choice 1, 2, 3 and 4), (Synchronized Parallelism 1, 2 and 3), (Hiding 1 and 2) and (Renaming 1 and 2). In these cases, $\omega''\backslash\omega' = \{\alpha\}$ and $\alpha = first(\alpha)$. Therefore, the lemma holds.

– (Sequential Composition 1) This case is trivial because $\omega''\backslash\omega' = \varnothing$.

□

Lemma 8. *Let $\mathcal{S}$ be a CSP specification, $\mathcal{G} = (N, E_c, E_l, E_s)$ the CSCFG associated with $\mathcal{S}$, and $s_i \xrightarrow{\Theta_i} s_{i+1}$, $i > 0$, a rewriting step of $\mathcal{D} = s_0 \longrightarrow \ldots \longrightarrow s_{n+1}$, $n \geqslant 0$, a* derivation *of $\mathcal{S}$ performed with the instrumented semantics, where $s_i = (Ctrl_\alpha, \omega_i)$ and $s_{i+1} = (Ctrl'_\varphi, \omega_{i+1})$. Then, $\exists\ \pi = n_j \mapsto^* n_k \in E_c$, $n_j, n_k \in N$, with $l(n_j) \in \omega_i$ and $l(n_k) = \mathit{first}(\alpha)$.*

Proof. Let us consider the rewriting step $s_{i-1} = (Ctrl_\zeta, \omega_{i-1}) \xrightarrow{\Theta_{i-1}} s_i = (Ctrl_\alpha, \omega_i)$.

If $\Theta_{i-1} = \varnothing$, rules (Process Call), (Parameterized Process Call), (Prefixing), (SKIP), (STOP), (Internal Choice 1 and 2), (Conditional Choice 1 and 2) and (Synchronized Parallelism 4) can be applied. In these cases, by Lemma 5 and Lemma 6, the lemma trivially holds.

If $\Theta_{i-1} \neq \varnothing$ and one of the rules (External Choice 1, 2, 3 and 4), (Synchronized Parallelism 1, 2, 3 and 4), (Sequential Composition 1), (Hiding 1, 2 and 3) or (Renaming 1, 2 and 3) is applied, we know by Lemma 5 and by Lemma 7 that $\mathit{first}(\zeta) \in \omega_i$ and that $\exists\ \pi = n_j \mapsto^* n_k \in E_c$, $n_j, n_k \in N$, $k \geqslant 1$, with $l(n_j) \in \omega_i$ and $l(n_k) = \mathit{first}(\alpha)$.

If rule (Sequential Composition 2) is applied, we know by Lemma 5 that $\mathit{first}(\zeta) \in \omega_i$ and by Definition 7 (using item 4 of Definition 3) and Definition 8, the context sensitive control of n_j can pass to n_k with $l(n_j) = \zeta \in \omega_i$, $lit(l(n_j)) =$; and $l(n_k) = first(\alpha)$, i.e., $\zeta \mapsto first(\alpha) \in E_c$.

□

The following lemma ensures that the CSCFG is complete: all possible derivations of a CSP specification $\mathcal{S}$ are represented in the CSCFG associated to $\mathcal{S}$.

Lemma 1. *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D} = s_0 \longrightarrow \ldots \longrightarrow s_{n+1}$, $n \geqslant 0$, a* derivation *of $\mathcal{S}$ performed with the instrumented semantics, where $s_0 = (rhs(\texttt{MAIN})_\alpha, \varnothing)$ and $s_{n+1} = (P_\varphi, \omega)$, and $\mathcal{G} = (N, E_c, E_l, E_s)$ the CSCFG associated with $\mathcal{S}$. Then, $\forall \gamma \in \omega :\ \exists\ \pi = n_1 \mapsto^* n_k \in E_c$, $n_1, n_k \in N$, $k \geqslant 1$, with $l(n_1) = first(\alpha)$ and $l(n_k) = \gamma$.*

Proof. We prove this lemma by induction on the length of the derivation $\mathcal{D}$. The base case happens when the length of $\mathcal{D}$ is one. The initial state is $s_0 = (rhs(\texttt{MAIN})_\alpha, \varnothing)$. The final state is $s_1 = (P_\varphi, \omega)$. In one rewriting step, one of the following rules must be applied:

- (Process Call) If $rhs(\texttt{MAIN}) = Q_\alpha$, this rule adds to ω the specification position α of P, and the control changes to $rhs(Q)_\varphi$, i.e., $s_1 = (rhs(Q)_\varphi, \{\alpha\})$. By Definition 8, the context sensitive control of n_0 can pass to n_1 with $l(n_0) =$ "$start$ $(\texttt{MAIN}, 0)$" and $l(n_1) = first((\texttt{MAIN}, \Lambda)) = \alpha$, i.e., "$start$ $(\texttt{MAIN}, 0)$" $\mapsto$ $\alpha \in E_c$. By Definition 3, $first((\texttt{MAIN}, \Lambda)) = \alpha = first(\alpha)$; by Definition 8, the context sensitive control of n_1 can pass to n_2 with $l(n_2) =$ "$start$ $(\texttt{Q}, 0)$" and by Lemma 6, $\exists\ \pi = n_1 \mapsto n_2 \mapsto n_3 \in E_c$ with $l(n_1) = first(\alpha)$ and $l(n_k) = first(\varphi)$, i.e., $\alpha \mapsto$ "$start$ $(\texttt{Q}, 0)$" $\mapsto first(\varphi) \in E_c$.

- (Parameterized Process Call) It is completely analogous to (Process Call).

- (Prefixing) If $rhs(\texttt{MAIN}) = a_\beta \rightarrow_\alpha P_\varphi$, this rule adds to ω the specification positions of the prefix and the prefixing operator, α and β respectively, and the control changes to P_φ, i.e., $s_1 = (P_\varphi, \{\alpha, \beta\})$. By Definition 8 and Definition 3, the context sensitive control of n_0 can pass to n_1 with $l(n_0) =$ "$start$ $(\texttt{MAIN}, 0)$" and $l(n_1) = first((\texttt{MAIN}, \Lambda)) = (\texttt{MAIN}, 1) = \beta$, i.e., "$start$ $(\texttt{MAIN}, 0)$" $\mapsto \beta \in E_c$. By Definition 7 (using item 2 of Definition 3) and Definition 8, the context sensitive control of n_1 can pass to n_2 with $l(n_1) = \beta$ and $l(n_2) = \alpha$, i.e., $\beta \mapsto \alpha \in E_c$. By Lemma 6, $\exists\ \pi = n_1 \mapsto n_2 \mapsto n_3 \in E_c$ with $l(n_1) = first(\alpha) = \beta$ and $l(n_3) = first((\texttt{MAIN}, 2)) = first(\varphi)$, i.e., $\beta \mapsto \alpha \mapsto first(\varphi) \in E_c$.

- (SKIP) If $rhs(\texttt{MAIN}) = \texttt{SKIP}_\alpha$, applying this rule the specification position α of SKIP is added to ω, the control changes to $\top$, i.e., $s_1 = (\top, \{\alpha\})$, and the derivation finishes. By Definition 8, the context sensitive control of n_0 can pass to n_1 with $l(n_0) =$ "$start$ $(\texttt{MAIN}, 0)$" and $l(n_1) = first((\texttt{MAIN}, \Lambda)) = \alpha$, i.e., "$start$ $(\texttt{MAIN}, 0)$" $\mapsto \alpha \in E_c$. And by Definition 7 and Definition 8, the context sensitive control of n_1 can pass to n_2 with $l(n_2) =$ "end $(\texttt{MAIN}, 0)$", i.e., $\alpha \mapsto$ "end $(\texttt{MAIN}, 0)$" $\in E_c$.

- (STOP) If $rhs(\texttt{MAIN}) = \texttt{STOP}_\alpha$, applying this rule the specification position α of STOP is added to ω, the control changes to $\bot$, i.e., $s_1 = (\bot, \{\alpha\})$, and the derivation finishes. By Definition 8, the context sensitive control of n_0 can pass to n_1 with $l(n_0) =$ "$start$ $(\texttt{MAIN}, 0)$" and $l(n_1) = first((\texttt{MAIN}, \Lambda)) = \alpha$, i.e., "$start$ $(\texttt{MAIN}, 0)$" $\mapsto \alpha \in E_c$.

- (Internal Choice 1 and 2) If $rhs(\texttt{MAIN}) = P_{\varphi_1} \sqcap_\alpha Q_{\varphi_2}$, with this rule the specification position of the choice operator α is added to ω, and one of the two processes P or Q is added to the control, i.e., $s_1 = (P_{\varphi_1}, \{\alpha\})$ or $s_1 = (Q_{\varphi_2}, \{\alpha\})$. By Definition 8, the context sensitive control of n_0 can pass to n_1 with $l(n_0) =$ "$start$ $(\texttt{MAIN}, 0)$" and $l(n_1) = first((\texttt{MAIN}, \Lambda)) = \alpha$, i.e., "$start$ $(\texttt{MAIN}, 0)$" $\mapsto \alpha \in E_c$. By Lemma 6, $\exists\ \pi = n_1 \mapsto n_2 \in E_c$ with $l(n_2) = \varphi = first((\texttt{MAIN}, 1)) = first(\varphi_1)$ and $\exists\ \pi = n_1 \mapsto n_3 \in E_c$ with $l(n_3) = first((\texttt{MAIN}, 2)) = first(\varphi_2)$, i.e., $\alpha \mapsto first(\varphi_1) \in E_c$ and $\alpha \mapsto first(\varphi_2) \in E_c$.

- (Conditional Choice 1 and 2) These rules are completely analogous to (Internal Choice 1 and 2).

– (External Choice 1, 2, 3 and 4) If $rhs(\texttt{MAIN}) = P \,\Box_\alpha Q$, with one of these rules the specification position of the choice operator α and the set of executed specification positions of process P or Q is added to ω. By Definition 8, the context sensitive control of n_0 can pass to n_1 with $l(n_0) =$ "$start\ (\texttt{MAIN}, 0)$" and $l(n_1) = first((\texttt{MAIN}, \Lambda)) = \alpha$, i.e., "$start\ (\texttt{MAIN}, 0)$" $\mapsto \alpha \in E_c$. By Lemma 7, $\forall \gamma \in \omega : \ \exists\ \pi = n_1 \mapsto^* n_k \in E_c$ with $l(n_k) = \gamma$.

– (Synchronized Parallelism 1 and 2) If $rhs(\texttt{MAIN}) = P \underset{X_\alpha}{\parallel} Q$, with one of these rules the specification position of the parallelism operator together with the specification positions executed of the corresponding process are added to ω. By Definition 8, the context sensitive control of n_0 can pass to n_1 with $l(n_0) =$ "$start\ (\texttt{MAIN}, 0)$" and $l(n_1) = first((\texttt{MAIN}, \Lambda)) = \alpha$, i.e., "$start\ (\texttt{MAIN}, 0)$" $\mapsto \alpha \in E_c$. By Lemma 7, $\forall \gamma \in \omega : \ \exists\ \pi = n_1 \mapsto^* n_k \in E_c$ with $l(n_k) = \gamma$.

– (Synchronized Parallelism 3) If $rhs(\texttt{MAIN}) = P \underset{X_\alpha}{\parallel} Q$, with this rule the specification position of the parallelism operator together with the specification positions executed of the two processes are added to ω. By Definition 8, the context sensitive control of n_0 can pass to n_1 with $l(n_0) =$ "$start\ (\texttt{MAIN}, 0)$" and $l(n_1) = first((\texttt{MAIN}, \Lambda)) = \alpha$, i.e., "$start\ (\texttt{MAIN}, 0)$" $\mapsto \alpha \in E_c$. By Lemma 7, $\forall \gamma \in \omega : \ \exists\ \pi = n_1 \mapsto^* n_k \in E_c$ with $l(n_k) = \gamma$.

– (Synchronized Parallelism 4) This rule does not add specification positions to ω.

– (Sequential Composition 1) If $rhs(\texttt{MAIN}) = P \ ;_\alpha Q$, this rule can be applied. The control changes to $P' \ ;_\alpha Q$ and the executed specification positions of P will be added to ω. By Definition 8, the context sensitive control of n_0 can pass to n_1 with $l(n_0) =$ "$start\ (\texttt{MAIN}, 0)$" and $l(n_1) = first((\texttt{MAIN}, \Lambda)) = first((\texttt{MAIN}, 1))$, i.e., "$start\ (\texttt{MAIN}, 0)$" $\mapsto first((\texttt{MAIN}, 1)) \in E_c$. By Lemma 7, $\forall \gamma \in \omega : \ \exists\ \pi = n_1 \mapsto^* n_k \in E_c$ with $l(n_k) = \gamma$.

– (Sequential Composition 2) For this rule to be applied, $rhs(\texttt{MAIN}) = SKIP_\beta \ ;_\alpha Q_\varphi$. The control changes to Q and $\omega = \{\alpha, \beta\}$, i.e., $s_1 = (Q_\varphi, \{\alpha, \beta\})$. By Definition 8, the context sensitive control of n_0 can pass to n_1 with $l(n_0) =$ "$start\ (\texttt{MAIN}, 0)$" and $l(n_1) = first((\texttt{MAIN}, \Lambda)) = first((\texttt{MAIN}, 1)) = \beta$, i.e., "$start\ (\texttt{MAIN}, 0)$" $\mapsto \beta \in E_c$. By Lemma 7, $n_1 \mapsto n_2 \in E_c$ with $l(n_2) = \alpha$. And by Definition 7 (using item 4 of Definition 3) and Definition 8, the context sensitive control of n_2 can pass to n_3 with $l(n_2) = \alpha$ and $l(n_3) = \varphi = first((\texttt{MAIN}, 2))$, i.e., $\alpha \mapsto \varphi \in E_c$.

– (Hiding 1 and 2) If $rhs(\texttt{MAIN}) = P \backslash_\alpha B$ and one of these rules is applied, ω is increased with the specification position α of the hiding operator and with the specification positions of the developed process P. By Definition 8, the context sensitive control of n_0 can pass to n_1 with $l(n_0) =$ "$start\ (\texttt{MAIN}, 0)$" and $l(n_1) = first((\texttt{MAIN}, \Lambda)) = \alpha$, i.e., "$start\ (\texttt{MAIN}, 0)$" $\mapsto \alpha \in E_c$. By Lemma 7, $\forall \gamma \in \omega : \ \exists\ \pi = n_1 \mapsto^* n_k \in E_c$ with $l(n_k) = \gamma$.

– (Hiding 3) For this rule to be applied, $rhs(\texttt{MAIN}) = SKIP_\beta \backslash_\alpha B$. The control changes to $\top$, $\omega = \{\alpha, \beta\}$, i.e., $s_1 = (\top, \{\alpha, \beta\})$, and the derivation

finishes. By Definition 8, the context sensitive control of n_0 can pass to n_1 with $l(n_0) =$ "$start$ $(\texttt{MAIN}, 0)$" and $l(n_1) = first((\texttt{MAIN}, \Lambda)) = \alpha$, i.e., "$start$ $(\texttt{MAIN}, 0)$" $\mapsto \alpha \in E_c$. By Lemma 7, $n_1 \mapsto n_2 \in E_c$ with $l(n_2) = first(\beta) = \beta$. And by Definition 7 (using item 6 of Definition 3) and Definition 8, the context sensitive control of n_2 can pass to n_3 with $l(n_3) =$ "end $(\texttt{MAIN}, \Lambda)$", and the context sensitive control of n_3 can pass to n_4 with $l(n_4) =$ "end $(\texttt{MAIN}, 0)$", i.e., $\beta \mapsto$ "end $(\texttt{MAIN}, 0)$" $\mapsto$ "end $(\texttt{MAIN}, 0)$" $\in E_c$,

– (Renaming 1 and 2) These rules are completely analogous to (Hiding 1 and 2).

– (Renaming 3) It is completely analogous to the rule (Hiding 3).

We assume as the induction hypothesis that the lemma holds in the i first rewriting steps of $\mathcal{D}$, and we prove that it also holds in the step $i + 1$. Let us consider $s_i = (Ctrl_\beta, \omega) \xrightarrow{\Theta_i} s_{i+1} = (Ctrl'_\delta, \omega')$. By the induction hypothesis we know that there exists a path from $first((\texttt{MAIN}, \Lambda))$ to all positions in ω. By Lemma 8, we know that there exists a path from a specification position in ω to $first(\beta)$. And by Lemma 7, we know that there exists a path from $first(\beta)$ to all positions in $\omega \backslash \omega'$. Therefore, the lemma holds. □

Lemma 3. *Let $\mathcal{S}$ be a CSP specification and let $\mathcal{G} = (N, E_c, E_l, E_s)$ be the CSCFG associated with $\mathcal{S}$ according to Definition 8. If $n \mapsto n' \in E_c$ then n must be executed before n' in all executions.*

Proof. We prove this lemma by induction on the length of a path $\pi_{\mathcal{S}tart} = n_1 \mapsto^* n_m$ in the CSCFG $\mathcal{G}$. Firstly, because all nodes in $\mathcal{S}tart(\mathcal{S})$ are not executable, we remove them from the path. Hence, we assume that for all $1 \leqslant j < m$ we have $l(n_j) \notin \mathcal{S}tart(\mathcal{S})$. We refer to this reduced path as π.

The base case happens when the length of π is one. The first node of the graph is always "$start$ $(\texttt{MAIN}, 0)$", hence, $\pi = n \mapsto n'$. Therefore, by Definition 8, $l(n) = first(\texttt{MAIN}, \Lambda)$.

In this situation, $lit(l(n))$ can be:

- If $lit(l(n)) = a$, with $a \in \Sigma$, then by Definition 3 (item 2), we have that $lit(l(n')) = \rightarrow$. In the semantics, the only rule applicable is Prefixing. Therefore, n must be executed before n'.

- If $lit(l(n)) = STOP$, by Definition 3 and Definition 8 there is no control from $STOP$. Therefore, $n' \notin N$ and thus this case is not possible.

- If $lit(l(n)) \in \{\sqcap, \Box, \nleftrightarrow, |||, ||\}$ then it is possible to apply Choice or Parallelism.

  – If we have a choice, by Definition 3 (item 1), $l(n') \in \{first((\texttt{MAIN}, 1)), first((\texttt{MAIN}, 2))\}$. In the semantics, the only rule applicable is a choice. Therefore, n must be executed before n'.

  – Analogously, if we have a parallelism, by Definition 3 (item 1), we have that $l(n') \in \{first((\texttt{MAIN}, 1)), first((\texttt{MAIN}, 2))\}$. In the semantics, the only rule applicable is a synchronized parallelism. Therefore, n must be executed before n'.

- If $lit(l(n)) \in \{\backslash, [\![]\!]\}$, then by Definition 3 (item 5), we have that $lit(l(n')) = first((\texttt{MAIN}, 1))$. In the semantics, the only rule applicable is Hiding or Renaming. Therefore, n must be executed before n'.
- If $lit(l(n)) = SKIP$ then two cases are possible:
  - by Definition 3 (item 3), $lit(l(n')) =$; because $last(l(n)) = l(n)$. In the semantics, the only rule applicable is Sequential Composition 2 and this necessarily implies that $SKIP$ is executed before. Thus, n must be executed before n'.
  - by Definition 8, we have in $\pi_{\mathcal{S}tart}$ that $lit(l(n')) =$ "end $(\texttt{MAIN}, 0)$" and in π, n' does not exist. In fact, in the semantics, the only rule applicable is SKIP that finishes the execution.
- If $lit(l(n)) = P$ with $P \in \mathcal{P}$ then, $l(n) = first(\texttt{MAIN}, \Lambda) = (\texttt{MAIN}, \Lambda)$, and by Definition 8, we have $\pi_{\mathcal{S}tart} =$ "$start$ $(\texttt{MAIN}, 0)$" $\mapsto (\texttt{MAIN}, \Lambda) \mapsto$ "$start$ $(\texttt{MAIN}, \Lambda)$" $\mapsto first((P, \Lambda))$. Therefore, $\pi = (\texttt{MAIN}, \Lambda) \mapsto first((P, \Lambda))$. In the semantics, the only rule applicable is Process Call that changes the root position in the control to (P, Λ). Then, by Lemma 5 we know that the next rewriting step will have $first((P, \Lambda)) \in \omega$.

We assume as the induction hypothesis that the lemma holds for a path π with length k. We prove that it also holds for a path with length $k+1$. Therefore, it is enough to prove that n_k must be executed before n_{k+1} in all executions with $n_k \mapsto n_{k+1} \in E_c$.

We analyze each possible case with respect to $lit(l(n_k))$. All cases are analogous to the base case except three:

- If $lit(l(n_k)) = \rightarrow$, with $l(n_k) = (M, w)$, then by Definition 3 (item 4), we have that $l(n_{k+1}) = first((M, w.2))$. In the semantics, the last rule applied was Prefixing, because it is the only rule that introduces $\rightarrow$. This rule puts in the control $(M, w.2)$. Therefore, by Lemma 5 we know that the next rewriting step will have $first((M, w.2)) \in \omega$. Therefore, n_k must be executed before n_{k+1}.
- If $lit(l(n_k)) =$; then it is completely analogous to the previous case but now the last rule applied is Sequential Composition 2.
- If $lit(l(n_k)) = SKIP$ then two cases are possible:
  - by Definition 3 (item 3), $lit(l(n_{k+1})) =$; because $last(l(n_k)) = l(n_k)$. In the semantics, the only rule applicable is Sequential Composition 2 and this necessarily implies that SKIP is executed before. Thus, n_k must be executed before n_{k+1}.
  - by Definition 8, we have in $\pi_{\mathcal{S}tart}$ that $lit(l(m+1)) =$ "end $(\texttt{MAIN}, 0)$" and in π, $m+1$ does not exist. In fact, in the semantics, the only rule applicable is SKIP that finishes the execution.

Therefore, the lemma is true. □

Lemma 2. *Let $\mathcal{S}$ be a CSP specification. Then, the execution of Algorithm 1 with $\mathcal{S}$ produces a graph $\mathcal{G}$ that is the CSCFG associated with $\mathcal{S}$ according to Definition 8.*

Proof. Let $\mathcal{G} = (N, E_c, E_l, E_s)$ the CSCFG associated with $\mathcal{S}$ according to Definition 8. And let $\mathcal{G}' = (N', E'_c, E'_l, E'_s)$ the output of Algorithm 1 with input $\mathcal{S}$. We now prove that $\mathcal{G} = \mathcal{G}'$.

In order to prove that both graphs are equivalent, we proceed by case analysis of function *buildGraph*. This is enough to prove the equivalence because this function is used to build the graph associated to the right hand side of all functions. Therefore, we have to prove not only that the graphs are equivalent for each case, but also that the returned values n_{first} and $Last$ are the expected ones so that recursion of this function also works and thus the produced graphs are correctly joined to form the final CSCFG. In all cases E_s will be equivalent to E'_s if the rest of the components of the graph are equivalent, because both are built using the same technique. We assume by induction hypothesis that, in all cases, the values returned by recursive calls are the expected ones. Let us study each case separately:

- Prefixing: In this case $P = X_\alpha \rightarrow_\beta Q$ and $X \in \{a, a?v, a!v\}$. We let $\beta = (M, w)$ thus $\alpha = (M, w.1)$ and the label of Q is $(M, w.2)$. Function *first* returns for this expression $(M, w.1)$. In the function it is represented by the fact that $n_{first} = n_\alpha$ (note that $l(n_\alpha) = \alpha = (M, w.1)$). Function $last$ will return the set $last(n2)$ where $l(n2) = (M, w.2)$. In the algorithm, the variable $Last$ is bound to $Last_1$, that is a set whose labels correspond to $last(n2)$. The nodes introduced by the algorithm are n_α and n_β. Their labels belong to $\mathcal{P}os(\mathcal{S})$, thus they are in N. The nodes in the set N_1 are also nodes from N. Hence, we can state that $N = N'$. This case introduces two control arcs plus the arcs introduced by the graph of Q. These two control arcs correspond to the second and fourth item of definition 3. The first is represented by $n_\alpha \mapsto n_\beta$, and the second by $n_\beta \mapsto n_{first1}$. Then, we have that $E_c = E'_c$. Finally, E'_l is equal to E_{l1}. Note that prefixing does not introduce loop arcs, so in this case we also have that $E_l = E'_l$.

- Choice and Parallelism: In this case $P = Q\ X_\alpha\ R$ and $X \in \{\sqcap, \Box, \not<\!\!>, |||, ||\}$. Let $\alpha = (M, w)$. Hence the labels of Q and R are $(M, w.1)$ and $(M, w.2)$ respectively. Function *first* returns (M, w) for this expression. In the function it is represented by the fact that $n_{first} = n_\alpha$. Function $last$ will return the set $last(n1) \cup last(n2)$, where $l(n1) = (M, w.1)$ and $l(n2) = (M, w.2)$, if none of these sets is empty or the operator is not a parallel operator neither an interleaving. Otherwise $last$ will return $\varnothing$. In the algorithm, the variable $Last$ is bounded to $Last_1 \cup Last_2$ (note that their labels will be all in last(n1) and last(n2)) or $\varnothing$ depending on the same condition. The nodes introduced by the algorithm are n_α and the nodes of sets N_1 and N_2. All these labels belong to $\mathcal{P}os(\mathcal{S})$, so we can state that $N = N'$. This case introduces two control arcs plus the arcs introduced by the graph of Q and R. These two control arcs correspond to the first item of definition 3. They are represented by $n_\alpha \mapsto first_1$ and $n_\alpha \mapsto first_2$. Then, we have that $E_c = E'_c$. Finally, E'_l is equal to $E_{l1} \cup E_{l2}$, hence $E_l = E'_l$.

- Sequential Composition: In this case $P = Q\ ;_\alpha\ R$. Let $\alpha = (M, w)$. Then the labels of Q and R are $(M, w.1)$ and $(M, w.2)$ respectively. Function

first returns for this expression $first((M, w.1))$. In the function it is represented by the fact that $n_{first} = n_{first1}$. Function *last* will return the set $last(n2)$, where $l(n2) = (M, w.2)$. In the algorithm, it is represented by the fact that the variable *Last* is bounded to $Last_2$ which their node labels are all in $last(n2)$. The nodes introduced by the algorithm are n_α and the sets of nodes N_1 and N_2. All these labels belong to $\mathcal{P}os(\mathcal{S})$, so, we can state that $N = N'$. This case introduces many control arcs plus the arcs introduced by the graph of Q and R. Concretely, it introduces the arc $n_\alpha \mapsto n_{first2}$ and the set E_{c3} where all the nodes belonging to $Last_1$ are joined to node n_α. The former corresponds to the fourth item of definition 3. The latter corresponds to the third item. Then, we have that $E_c = E'_c$. Finally, E'_l is equal to $E_{l1} \cup E_{l2}$, so $E_l = E'_l$.

- Hiding and Renaming: In this case $P = Q\ X_\alpha$ and $X \in \{\backslash, [\![]\!]\}$. Let $\alpha = (M, w)$ then the label of Q is $(M, w.1)$. Function *first* returns for this expression (M, w). In the function, it is represented by the fact that $n_{first} = n_\alpha$ (note that $l(n_\alpha) = \alpha = (M, w)$). Function *last* will return the set $last(n1)$, where $l(n1) = (M, w.1)$. In the algorithm, the variable *Last* is bounded to $Last_1$ and all their node labels are in $last(n1)$. The nodes introduced by the algorithm are n_α, n_{end} and the nodes of sets N_1. All these labels belong to $\mathcal{P}os(\mathcal{S})$, except n_{end} that belongs to $\mathcal{S}tart(\mathcal{S})$. Thus, we can state that $N = N'$. This case introduces the control arc $n_\alpha \mapsto n_first1$, the arcs of E_{c2} plus the arcs introduced by the graph of Q. The single arc corresponds to the fifth item of definition 3, and the edges in E_{c2} correspond to the sixth item of the same definition. Then, we have that $E_c = E'_c$. Finally, E'_l is equal to $E_{l1} \cup E_{l2}$, so $E_l = E'_l$.

- SKIP and STOP: In this case $P = Q\ X_\alpha$ and $X \in \{SKIP, STOP\}$. Let $\alpha = (M, w)$. Function *first* returns for this expression (M, w). In the function it is represented by the fact that $n_{first} = n_\alpha$ (note that $l(n_\alpha) = \alpha = (M, w)$). Function *last* will return the set $\{(M, w)\}$ if $lit(l(n)) = SKIP$ or $\varnothing$ if $lit(l(n)) = STOP$. In the algorithm, the variable *Last* is bounded to $\{n_\alpha\}$ or $\varnothing$ with the same conditions. The only node introduced by the algorithm is n_α that belongs to $\mathcal{P}os(\mathcal{S})$. Thus, we can state that $N = N'$. This case does not introduce any control arc. Then, we have that $E_c = E'_c$, because with only one node it is not possible to have control flow. Finally, E'_l is equal to $\varnothing$, so, as it happens in the previous case, $E_l = E'_l$.

- (Parameterized) Process Call: In this case $P = X_\alpha$. Let $\alpha = (M, w)$. Function *first* returns (M, w) for this expression. In the function it is represented by the fact that $n_{first} = n_\alpha$ (note that $l(n_\alpha) = \alpha = (M, w)$). Function *last* will return $\varnothing$ or $\{$"$end(M, w)$"$\}$ depending on whether a node with label "$start(M, w)$" is in $\mathcal{C}on(n)$ or not respectively. This is the same condition that we find in the conditional clause of the algorithm, thus we have a total correspondence. The graph components also depend in this condition, so we are going to distinguish between two cases. First, when the start node is in the context, and second when it is not.

  - The only node introduced by the algorithm is n_α that belongs to $\mathcal{P}os(\mathcal{S})$. Thus, we can state that $N = N'$. This case does not

introduce any control arc. Then, we have that $E_c = E'_c$. Finally, E'_l is equal to a set with a unique loop arc from n_α to n_{ctx} (which is the node that makes the condition hold). This arc is the same as the one introduced in the same conditions of definition 8, so $E_l = E'_l$.

– The nodes introduced by the algorithm are n_α, that belongs to $\mathcal{P}os(\mathcal{S})$, and n_{start} and n_{end} that belongs to $\mathcal{S}tart(\mathcal{S})$. These nodes plus set N_1 form N', so we can state that $N = N'$. The set E'_c if formed by the union of set E_{c1} from the graph of the right-hand side of process X, and set E_{c2}. The latter represents the special control flow stated in the second item of definition 8. Then, we have that $E_c = E'_c$. Finally, E'_l is equal to E_{l1}, so $E_l = E'_l$.

□

Lemma 4. (Finiteness) *Given a specification $\mathcal{S}$, its associated CSCFG is finite.*

Proof. We show first that there does not exist infinite unfolding in a CSCFG. Firstly, the same start process node only appears twice in the same control loop-free path if it belongs to a process which is called from different process calls (i.e., with different specification positions) as it is ensured by Definition 8. Therefore, the number of repeated nodes in the same control loop-free path is limited by the number of different process calls appearing in the program. Moreover, the number of terms in the specification is finite and thus there is a finite number of different process calls. In addition, every process call has only one outgoing arc as it is ensured by the third property of Definition 8. Therefore, the number of paths is finite and the size of every path of the CSCFG is limited.

□

Theorem 8. (Termination of MEB) *The MEB analysis performed by Algorithm 2 terminates.*

Proof. First, we know that N is finite, and thus *blseeds* is also finite because $blseeds \subset N$. Therefore, the first loop (4) always terminates because it is repeated while new nodes are added to *blacklist*; and the number of possible insertions is finite because N is finite. We can ensure that the second loop also terminates due to the invariant $pending \cap Meb = \varnothing$ which is always true at the end of the loop (9). Then, because *Meb* increases in every iteration (7) and the size of N is finite, *pending* will eventually become empty and the loop will terminate.

□

Theorem 10. (Termination of CEB) *The CEB analysis performed by Algorithm 3 terminates.*

Proof. Firstly, the algorithm starts with a call to the function *buildMeb*. By Lemma 8, this call always terminates. Then, the only loops that could cause non-termination are the loop containing sentences (4) to (10) and the loop containing sentences (13) to (18). The first loop is repeated until no new nodes are added to the sets *loopnodes* or *candidates*. We know that *loopnodes* never decreases in the loop; moreover, sentence (4) ensures that $m' \notin loopnodes$, therefore, the number of nodes added to *loopnodes* is finite because the number of nodes in N is finite. Similarly, *candidates* only have a finite number of

insertions, and once a node is added to *candidates* it can be removed, but never inserted again because *loopnodes* never decreases. The second loop is analogous to the first one. Therefore, we can ensure that it always terminates by showing that *Ceb* is increased in every iteration with nodes of *pending* that leave *pending* when they are inserted into *Ceb*, see (14) and (17). And, moreover, *pending* can only be increased a limited number of times because it is always increased with nodes which are the successor of a node in *pending* following control arcs. Therefore, because the CSCFG is a tree if we only consider control arcs and N is finite, the size of every branch is finite, and thus, the loop always terminates. □

Theorem 9. (Completeness of MEB) *Let $\mathcal{S}$ be a specification, $\mathcal{C}$ a slicing criterion for $\mathcal{S}$, and let $\mathcal{MEB}$ be the MEB slice of $\mathcal{S}$ with respect to $\mathcal{C}$. Then, $\mathcal{MEB} \subseteq MEB(\mathcal{S}, \mathcal{C})$.*

Proof. (sketch) First, we prove that $MEB(\mathcal{S}, \mathcal{C})$ considers all possible executions of the slicing criterion as $\mathcal{MEB}$ does. This depends on function *nodes*, that considers only the first occurrences (starting from `MAIN` and proceeding forwards) of the slicing criterion. Then, it is equivalent to consider the first time that the slicing criterion belongs to ω, which is the stopping condition in the $\mathcal{MEB}$ construction process. Second, as the slicing criterion could happen in different executions, we have to construct a relation between them in order to build the final result. In the case of $\mathcal{MEB}$ the intersection of ω for each execution is considered. However, $MEB(\mathcal{S}, \mathcal{C})$ considers the intersection of the specification positions of each node belonging to *buildMeb*(n) where n is a slicing criterion's node. So we have to prove that the result of *buildMeb*(n) will return all (and maybe more) the nodes (and consequently specification positions) that have been executed before it. Function *buildMeb* in step (2) selects all the nodes from `MAIN` to the given node n and to those nodes which are synchronized with it. The rest of the nodes that will be appended in the rest of steps depends mainly on sets *pending* and *sync*. These sets will be formed by those nodes that are synchronized with nodes in set *Meb* or loops which contain at least one node synchronized with a node in *Meb*. Then all possible executed nodes are belonging to *Meb* at the end. The only problem that could arise happens when some nodes are not considered and they are included in the *blacklist*, the set of discarded nodes. However, all the nodes in this set are correctly discarded, because it adds first the given node n and the branches of choices or interleaving which do not reach a node in *Meb*; then, it discards iteratively the nodes under these ones and all that are synchronized with them only if all the other nodes synchronized are also discarded. Therefore, we can conclude that the result will be a superset of $\mathcal{MEB}$. □

Theorem 11. (Completeness of CEB) *Let $\mathcal{S}$ be a specification, $\mathcal{C}$ a slicing criterion for $\mathcal{S}$, and let $\mathcal{CEB}$ be the CEB slice of $\mathcal{S}$ with respect to $\mathcal{C}$. Then, $\mathcal{CEB} \subseteq CEB(\mathcal{S}, \mathcal{C})$.*

Proof. (sketch) The first part of the proof is completely analogous to the previous one. The rest of the proof concerns function *buildCeb*. In its first step, a call to function *buildMeb* is made. Consequently all those parts that must be executed before node n will form the initial set for *Ceb*. Then, we have to prove that the rest of steps collects those nodes that could also be executed before the

given node n. This group only depends on loops that finish in a node that is in *Ceb*. Then, in step (2) the set *loopnodes* is initialized with the children of the choices in *Ceb* that are not in *Ceb*. After this, an iterative process proceeds forward adding nodes if they could be executed, i.e. if it has not synchronization arcs; or in case it has, their synchronized nodes are in *Ceb* or in *candidates* (a set of nodes waiting for acceptance). In this way, it is assured that only those nodes that could be executed before n are added to *Ceb* in step (11). Finally, the same checking idea is applied to the nodes that are under nodes in *Ceb* (but n), adding iteratively more nodes if the conditions are fulfilled. With this last step, the rest of specification positions that could be executed (those belonging to other threads of execution) is safely added to the set *Ceb*. Then, we can conclude that $CEB(\mathcal{S},\mathcal{C})$ is a superset of $\mathcal{CEB}$. □

Graph Generation to Statically Represent CSP Processes[☆]

Marisa Llorens[a], Javier Oliver[a], Josep Silva[a], Salvador Tamarit[a]

[a] *Universitat Politècnica de València, Camino de Vera S/N, E-46022 Valencia, Spain*

Abstract

The CSP language allows the specification and verification of complex concurrent systems. Many analyses for CSP exist that have been successfully applied in different industrial projects. However, the cost of the analyses performed is usually very high, and sometimes prohibitive, due to the complexity imposed by the non-deterministic execution order of processes and to the restrictions imposed on this order by synchronizations. In this work, we define a data structure that allows us to statically simplify a specification before the analyses. This simplification can drastically reduce the time needed by many CSP analyses. We also introduce an algorithm able to automatically generate this data structure from a CSP specification. The algorithm has been proved correct and its implementation for the CSP's animator ProB is publicly available.

1. Introduction

The *Communicating Sequential Processes* (CSP) [3, 13] language allows us to specify complex systems with multiple interacting processes. The study and transformation of such systems often implies different analyses (e.g., deadlock analysis [5], reliability analysis [4], refinement checking [12], etc.) which are often based on a data structure able to represent all computations of a specification.

Recently, a new data structure called *Context-sensitive Synchronized Control-Flow Graph* (CSCFG) has been proposed [7]. This data structure is a graph that allows us to finitely represent possibly infinite computations, and it is particularly interesting because it takes into account the context of process calls, and thus it allows us to produce analyses that are very precise. In particular, some analyses (see, e.g., [8, 9]) use the CSCFG to simplify a specification with respect to some term by discarding those parts of the specification that cannot

[☆]This work has been partially supported by the Spanish *Ministerio de Ciencia e Innovación* under grant TIN2008-06622-C03-02, by the *Generalitat Valenciana* under grant ACOMP/2010/042, and by the *Universidad Politécnica de Valencia* (Program PAID-06-08). Salvador Tamarit was partially supported by the Spanish MICINN under FPI grant BES-2009-015019.

Email addresses: `mllorens@dsic.upv.es` (Marisa Llorens), `fjoliver@dsic.upv.es` (Javier Oliver), `jsilva@dsic.upv.es` (Josep Silva), `stamarit@dsic.upv.es` (Salvador Tamarit)

be executed before the term and thus they cannot influence it. This simplification is automatic and thus it is very useful as a preprocessing stage of other analyses.

However, computing the CSCFG is a complex task due to the non-deterministic execution of processes, due to deadlocks, due to non-terminating processes and mainly due to synchronizations. This is the reason why there does not exist any correctness result which formally relates the CSCFG of a specification to its execution. This result is needed to prove important properties (such as correctness and completeness) of the techniques based on the CSCFG.

In this work, we formally define the CSCFG and a technique to produce the CSCFG of a given CSP specification. Roughly, we instrument the CSP standard semantics (Chapter 7 in [13]) in such a way that the execution of the instrumented semantics produces as a side-effect the portion of the CSCFG associated with the performed computation. Then, we define an algorithm which uses the instrumented semantics to build the complete CSCFG associated with a CSP specification. This algorithm executes the semantics several times to explore all possible computations of the specification, producing incrementally the final CSCFG.

2. The Syntax and Semantics of CSP

In order to make the paper self-contained, this section recalls CSP's syntax and semantics [3, 13]. For concretion, and to facilitate the understanding of the following definitions and algorithm, we have selected a subset of CSP that is sufficiently expressive to illustrate the method, and it contains the most important operators that produce the challenging problems such as deadlocks, non-determinism and parallel execution.

We use the following domains: process names ($M, N \ldots \in Names$), processes ($P, Q \ldots \in Procs$) and events ($a, b \ldots \in \Sigma$). A CSP specification is a finite set of process definitions $N = P$ with $P = M \mid a \rightarrow P \mid P \sqcap Q \mid P \,\Box\, Q \mid P \underset{X \subseteq \Sigma}{||} Q \mid STOP$. Therefore, processes can be a call to another process or a combination of the following operators:

Prefixing ($a \rightarrow P$) Event a must happen before process P.

Internal choice ($P \sqcap Q$) The system chooses non-deterministically to execute one of the two processes P or Q.

External choice ($P \,\Box\, Q$) It is identical to internal choice but the choice comes from outside the system (e.g., the user).

Synchronized parallelism ($P \underset{X \subseteq \Sigma}{||} Q$) Both processes are executed in parallel with a set X of synchronized events. In absence of synchronizations both processes can execute in any order. Whenever a synchronized event $a \in X$ happens in one of the processes, it must also happen in the other at the same time. Whenever the set of synchronized events is not specified, it is assumed that processes are synchronized in all common events. A particular case of parallel execution is *interleaving* (represented by $|||$) where no synchronizations exist (i.e., $X = \emptyset$).

Stop ($STOP$) Synonym of deadlock: It finishes the current process.

We now recall the standard operational semantics of CSP as defined by Roscoe [13]. It is presented in Fig. 1 as a logical inference system. A *state* of

the semantics is a process to be evaluated called the *control*. In the following, we assume that the system starts with an initial state `MAIN`, and the rules of the semantics are used to infer how this state evolves. When no rules can be applied to the current state, the computation finishes. The rules of the semantics change the states of the computation due to the occurrence of events. The set of possible events is $\Sigma^\tau = \Sigma \cup \{\tau\}$. Events in Σ are visible from the external environment, and can only happen with its co-operation (e.g., actions of the user). Event τ is an internal event that cannot be observed from outside the system and it happens automatically as defined by the semantics. In order to perform computations, we construct an initial state and (non-deterministically) apply the rules of Fig. 1.

(Process Call) $\dfrac{}{N \xrightarrow{\tau} rhs(N)}$ (Prefixing) $\dfrac{}{(a \to P) \xrightarrow{a} P}$ (Internal Choice 1) $\dfrac{}{(P \sqcap Q) \xrightarrow{\tau} P}$ (Internal Choice 2) $\dfrac{}{(P \sqcap Q) \xrightarrow{\tau} Q}$

(External Choice 1) $\dfrac{P \xrightarrow{\tau} P'}{(P \,\Box\, Q) \xrightarrow{\tau} (P' \,\Box\, Q)}$ (External Choice 2) $\dfrac{Q \xrightarrow{\tau} Q'}{(P \,\Box\, Q) \xrightarrow{\tau} (P \,\Box\, Q')}$ (External Choice 3) $\dfrac{P \xrightarrow{e} P'}{(P \,\Box\, Q) \xrightarrow{e} P'}\ e \in \Sigma$ (External Choice 4) $\dfrac{Q \xrightarrow{e} Q'}{(P \,\Box\, Q) \xrightarrow{e} Q'}\ e \in \Sigma$

(Synchronized Parallelism 1) $\dfrac{P \xrightarrow{e} P'}{(P \underset{X}{||} Q) \xrightarrow{e} (P' \underset{X}{||} Q)}\ e \in \Sigma^\tau \backslash X$ (Synchronized Parallelism 2) $\dfrac{Q \xrightarrow{e} Q'}{(P \underset{X}{||} Q) \xrightarrow{e} (P \underset{X}{||} Q')}\ e \in \Sigma^\tau \backslash X$ (Synchronized Parallelism 3) $\dfrac{P \xrightarrow{e} P' \quad Q \xrightarrow{e} Q'}{(P \underset{X}{||} Q) \xrightarrow{e} (P' \underset{X}{||} Q')}\ e \in X$

Figure 1: CSP's operational semantics

3. Context-sensitive Synchronized Control-Flow Graphs

The CSCFG was proposed in [7, 9] as a data structure able to finitely represent all possible (often infinite) computations of a CSP specification. This data structure is particularly useful to simplify a CSP specification before its static analysis. The simplification of industrial CSP specifications allows us to drastically reduce the time needed to perform expensive analyses such as model checking. Algorithms to construct CSCFGs have been implemented [8] and integrated into the most advanced CSP environment ProB [6]. In this section we introduce a new formalization of the CSCFG that directly relates the graph construction to the control-flow of the computations it represents.

A CSCFG is formed by the sequence of expressions that are evaluated during an execution. These expressions are conveniently connected to form a graph. In addition, the source position (in the specification) of each literal (i.e., events, operators and process names) is also included in the CSCFG. This is very useful because it provides the CSCFG with the ability to determine what parts of the source code have been executed and in what order. The inclusion of source positions in the CSCFG implies an additional level of complexity in the semantics, but the benefits of providing the CSCFG with this additional information are clear and, for some applications, essential. Therefore, we use labels (that we call *specification positions*) to identify each literal in a specification which roughly corresponds to nodes in the CSP specification's abstract syntax tree. We define a function $\mathcal{P}os$ to obtain the specification position of an element of a CSP specification and it is defined over nodes of an abstract syntax tree for a CSP specification. Formally,

Definition 1. (Specification position) A *specification position* is a pair (N, w) where $N \in \mathcal{N}$ and w is a sequence of natural numbers (we use Λ to denote the empty sequence). We let $\mathcal{P}os(o)$ denote the specification position of an expression o. Each process definition $N = P$ of a CSP specification is labelled with specification positions. The specification position of its left-hand side is $\mathcal{P}os(N) = (N, 0)$. The right-hand side (abbrev. *rhs*) is labelled with the call $\texttt{AddSpPos}(P, (N, \Lambda))$; where function `AddSpPos` is defined as follows:

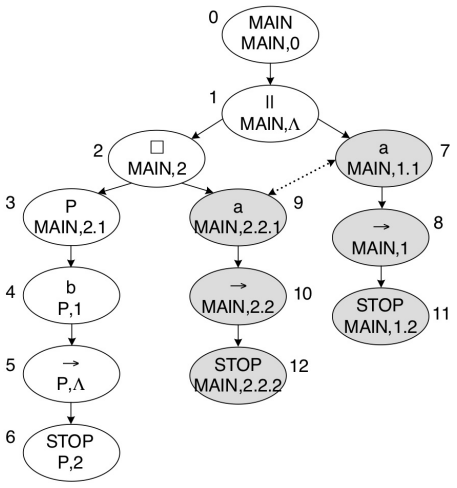
$$\texttt{AddSpPos}(P,(N,w)) = \begin{cases} P_{(N,w)} & \text{if } P \in \mathcal{N} \\ STOP_{(N,w)} & \text{if } P = STOP \\ a_{(N,w.1)} \rightarrow_{(N,w)} \texttt{AddSpPos}(Q,(N,w.2)) & \text{if } P = a \rightarrow Q \\ \texttt{AddSpPos}(Q,(N,w.1))\ op_{(N,w)}\ \texttt{AddSpPos}(R,(N,w.2)) & \text{if } P = Q\ op\ R\ \ \forall\ op \in \{\sqcap, \Box, ||\} \end{cases}$$

We often use $\mathcal{P}os(\mathcal{S})$ to denote a set with all positions in a specification $\mathcal{S}$.

Example 1. *Consider the CSP specification in Fig. 2(a) where literals are labelled with their associated specification positions (they are underlined) so that labels are unique.*

$$\texttt{MAIN}_{\underline{(\texttt{MAIN},0)}} = (\texttt{a}_{\underline{(\texttt{MAIN},1.1)}} \rightarrow {}_{\underline{(\texttt{MAIN},1)}}\texttt{STOP}_{\underline{(\texttt{MAIN},1.2)}}) \underset{\{\texttt{a}\}}{\|}{}_{\underline{(\texttt{MAIN},\Lambda)}}$$
$$(\texttt{P}_{\underline{(\texttt{MAIN},2.1)}} \Box_{\underline{(\texttt{MAIN},2)}} (\texttt{a}_{\underline{(\texttt{MAIN},2.2.1)}} \rightarrow {}_{\underline{(\texttt{MAIN},2.2)}}\texttt{STOP}_{\underline{(\texttt{MAIN},2.2.2)}}))$$
$$\texttt{P}_{\underline{(\texttt{P},0)}} = \texttt{b}_{\underline{(\texttt{P},1)}} \rightarrow {}_{\underline{(\texttt{P},\Lambda)}}\texttt{SKIP}_{\underline{(\texttt{P},2)}}$$

(a) CSP specification



(b) CSCFG

Figure 2: CSP specification and its associated CSCFG

In the following, specification positions will be represented with greek letters $(\alpha, \beta, \ldots)$ and we will often use indistinguishably an expression and its associ-

ated specification position when it is clear from the context (e.g., in Example 1 we will refer to $(\mathtt{P}, 1)$ as $\mathtt{b}$).

In order to introduce the definition of CSCFG, we need first to define the concepts of *control-flow*, *path* and *context*.

Definition 2. (Control-flow) Given a CSP specification $\mathcal{S}$, the *control-flow* is a transitive relation between the specification positions of $\mathcal{S}$. Given two specification positions α, β in $\mathcal{S}$, we say that the *control* of α *can pass* to β iff

i) $\alpha = N \ \wedge \ \beta = \mathit{first}((N, \Lambda))$ with $N = rhs(N) \in \mathcal{S}$
ii) $\alpha \in \{\sqcap, \Box, ||\} \ \wedge \ \beta \in \{\mathit{first}(\alpha.1), \mathit{first}(\alpha.2)\}$
iii) $\alpha = \beta.1 \ \wedge \ \beta = \ \rightarrow$
iv) $\alpha = \ \rightarrow \ \wedge \ \beta = \mathit{first}(\alpha.2)$

where $\mathit{first}(\alpha)$ is defined as follows: $\mathit{first}(\alpha) = \begin{cases} \alpha.1 & \text{if } \alpha = \ \rightarrow \\ \alpha & \text{otherwise} \end{cases}$

We say that a specification position α is *executable* in $\mathcal{S}$ iff the control can pass from the initial state (i.e., $\mathtt{MAIN}$) to α.

For instance, in Example 1, the control can pass from $(\mathtt{MAIN}, 2.1)$ to $(\mathtt{P}, 1)$ due to rule i), from $(\mathtt{MAIN}, 2)$ to $(\mathtt{MAIN}, 2.1)$ and $(\mathtt{MAIN}, 2.2.1)$ due to rule ii), from $(\mathtt{MAIN}, 2.2.1)$ to $(\mathtt{MAIN}, 2.2)$ due to rule iii), and from $(\mathtt{MAIN}, 2.2)$ to $(\mathtt{MAIN}, 2.2.2)$ due to rule iv).

As we will work with graphs whose nodes are labelled with positions, we use $l(n)$ to refer to the label of node n.

Definition 3. (Path) Given a labelled graph $\mathcal{G} = (N, E)$, a *path* between two nodes n_1, $m \in N$, $Path(n_1, m)$, is a sequence $n_1, \ldots, n_k$ such that $n_k \mapsto m \in E$ and for all $1 \leq i < k$ we have $n_i \mapsto n_{i+1} \in E$. The path is *loop-free* if for all $i \neq j$ we have $n_i \neq n_j$.

Definition 4. (Context) Given a labelled graph $\mathcal{G} = (N, E)$ and a node $n \in N$, the *context* of n, $\mathcal{C}on(n) = \{m \mid l(m) = M$ with $(M = P) \in \mathcal{S}$ and there exists a loop-free path $m \mapsto^* n\}$.

Intuitively speaking, the context of a node represents the set of processes in which a particular node is being executed. This is represented by the set of process calls in the computation that were done before the specified node. For instance, the CSCFG associated with the specification in Example 1 is shown in Fig. 2(b). In this graph we have that $\mathcal{C}on(4) = \{0, 3\}$, i.e., $\mathtt{b}$ is being executed after having called processes $\mathtt{MAIN}$ and $\mathtt{P}$. Note that focussing on a process call node we can use the context to identify loops; i.e., we have a loop whenever $n \in \mathcal{C}on(m)$ with $l(n) = l(m) \in Names$. Note also that the CSCFG is unique for a given CSP specification [9].

Definition 5. (Context-sensitive Synchronized Control-Flow Graph) Given a CSP specification $\mathcal{S}$, its *Context-sensitive Synchronized Control-Flow Graph* (CSCFG) is a labelled directed graph $\mathcal{G} = (N, E_c, E_l, E_s)$ where N is a set of nodes such that $\forall \ n \in N.\ l(n) \in \mathcal{P}os(\mathcal{S})$ and $l(n)$ is executable in $\mathcal{S}$; and edges are divided into three groups: *control-flow edges* (E_c), *loop edges* (E_l) and *synchronization edges* (E_s).

- E_c is a set of one-way edges (denoted with $\mapsto$) representing the possible control-flow between two nodes. Control edges do not form loops. The root of the tree formed by E_c is the position of the initial call to `MAIN`.
- E_l is a set of one-way edges (denoted with $\rightsquigarrow$) such that $(n_1 \rightsquigarrow n_2) \in E_l$ iff $l(n_1)$ and $l(n_2)$ are (possibly different) process calls that refer to the same process $M \in \mathcal{N}$ and $n_2 \in \mathcal{C}on(n_1)$.
- E_s is a set of two-way edges (denoted with $\leftrightarrow$) representing the possible synchronization of two event nodes ($l(n) \in \Sigma$).
- Given a CSCFG, every node labelled (M, Λ) has one and only one incoming edge in E_c; and every process call node has one and only one outgoing edge which belongs to either E_c or E_l.

Example 2. *Consider again the specification of Example 1, shown in Fig. 2(a), and its associated CSCFG, shown in Fig. 2(b). For the time being, the reader can ignore the numbering and color of the nodes; they will be explained in Section 4. Each process call is connected to a subgraph which contains the right-hand side of the called process. For convenience, in this example there are no loop edges;*[1] *there are control-flow edges and one synchronization edge between nodes* $(\texttt{MAIN}, \texttt{2.2.1})$ *and* $(\texttt{MAIN}, \texttt{1.1})$ *representing the synchronization of event* `a`.

Note that the CSCFG shows the exact processes that have been evaluated with an explicit causality relation; and, in addition, it shows the specification positions that have been evaluated and in what order. Therefore, it is not only useful as a program comprehension tool, but it can be used for program simplification. For instance, with a simple backwards traversal from `a`*, the CSCFG reveals that the only part of the code that can be executed before* `a` *is the underlined part:*

$$\underline{\texttt{MAIN}} = (\underline{\texttt{a}} \rightarrow \texttt{STOP}) \underset{\underline{\{\texttt{a}\}}}{\parallel} (\texttt{P} \,\underline{\Box}\, (\underline{\texttt{a}} \rightarrow \texttt{STOP}))$$
$$\texttt{P} = \texttt{b} \rightarrow \texttt{STOP}$$

Hence, the specification can be significantly simplified for those analyses focussing on the occurrence of event `a`.

4. An Algorithm to Generate the CSCFG

This section introduces an algorithm able to generate the CSCFG associated with a CSP specification. The algorithm uses an instrumented operational semantics of CSP which (i) generates as a side-effect the CSCFG associated with the computation performed with the semantics; (ii) it controls that no infinite loops are executed; and (iii) it ensures that the execution is deterministic.

Algorithm 1 controls that the semantics is executed repeatedly in order to deterministically execute all possible computations—of the original (non-deterministic) specification—and the CSCFG for the whole specification is constructed incrementally with each execution of the semantics. The key point of the algorithm is the use of a stack that records the actions that can be performed by the semantics. In particular, the stack contains tuples of the form

[1] We refer the reader to [10] where an example with loop edges is discussed.

$(rule, rules)$ where $rule$ indicates the rule that must be selected by the semantics in the next execution step, and $rules$ is a set with the other possible rules that can be selected. The algorithm uses the stack to prepare each execution of the semantics indicating the rules that must be applied at each step. For this, function `UpdStack` is used; it basically avoids to repeat the same computation with the semantics. When the semantics finishes, the algorithm prepares a new execution of the semantics with an updated stack. This is repeated until all possible computations are explored (i.e., until the stack is empty).

The standard operational semantics of CSP [13] can be non-terminating due to infinite computations. Therefore, the instrumentation of the semantics incorporates a loop-checking mechanism to ensure termination.

Algorithm 1 General Algorithm

Build the initial state of the semantics: $state = (\texttt{MAIN}_{(\texttt{MAIN},0)}, \emptyset, \bullet, (\emptyset, \emptyset), \emptyset, \emptyset)$
repeat
  repeat
    Run the rules of the instrumented semantics with the state $state$
  until no more rules can be applied
  Get the new state: $state = (\_, G, \_, (\emptyset, S_0), \_, \zeta)$
  $state = (\texttt{MAIN}_{(\texttt{MAIN},0)}, G, \bullet, (\texttt{UpdStack}(S_0), \emptyset), \emptyset, \emptyset)$
until $\texttt{UpdStack}(S_0) = \emptyset$
return G
where function `UpdStack` is defined as follows:

$$\texttt{UpdStack}(S) = \begin{cases} (rule, rules \backslash \{rule\}) : S' & \text{if } S = (\_, rules) : S' \text{ and } rule \in rules \\ \texttt{UpdStack}(S') & \text{if } S = (\_, \emptyset) : S' \\ \emptyset & \text{if } S = \emptyset \end{cases}$$

The instrumented semantics used by Algorithm 1 is shown in Fig. 3. It is an operational semantics where we assume that every literal in the specification has been labelled with its specification position (denoted by a subscript, e.g., P_α). In this semantics, a *state* is a tuple $(P, G, m, (S, S_0), \Delta, \zeta)$, where P is the process to be evaluated (the *control*), G is a directed graph (i.e., the CSCFG constructed so far), m is a numeric reference to the current node in G, (S, S_0) is a tuple with two stacks (where the empty stack is denoted by $\emptyset$) that contains the rules to apply and the rules applied so far, Δ is a set of references to nodes used to draw synchronizations in G and ζ is a graph like G, but it only contains the part of the graph generated for the current computation, and it is used to detect loops. The basic idea of the graph construction is to record the current control with a fresh reference[2] n by connecting it to its parent m. We use the notation $G[n \overset{m}{\mapsto} \alpha]$ either to introduce a node in G or as a condition on G (i.e., G contains node n). This node has reference n, is labelled with specification position α and its parent is m. The edge introduced can be a control, a synchronization or a loop edge. This notation is very convenient because it allows us to add nodes to G, but also to extract information from G. For instance, with $G[3 \overset{m}{\mapsto} \alpha]$ we can know the parent of node 3 (the value of m), and the specification position

[2]We assume that fresh references are numeric and generated incrementally.

of node 3 (the value of α).

Note that the initial state for the semantics used by Algorithm 1 has $\texttt{MAIN}_{(\texttt{MAIN},0)}$ in the control. This initial call to `MAIN` does not appear in the specification, thus we label it with a special specification position $(\texttt{MAIN}, 0)$ which is the root of the CSCFG (see Fig. 2(b)). Note that we use $\bullet$ as a reference in the initial state. The first node added to the CSCFG (i.e., the root) will have parent reference $\bullet$. Therefore, here $\bullet$ denotes the empty reference because the root of the CSCFG has no parent.

An explanation for each rule of the semantics follows.

(Process Call)

$$\frac{}{(N_\alpha, G, m, (S, S_0), \Delta, \zeta) \xrightarrow{\tau} (P', G', n, (S, S_0), \emptyset, \zeta')}$$

$$(P', G', \zeta') \;=\; \texttt{LoopCheck}(N, n, G[n \overset{m}{\mapsto} \alpha], \zeta \cup \{n \overset{m}{\mapsto} \alpha\})$$

(Prefixing)

$$\frac{}{(a_\alpha \rightarrow_\beta P, G, m, (S, S_0), \Delta, \zeta) \xrightarrow{a} (P, G[n \overset{m}{\mapsto} \alpha, o \overset{n}{\mapsto} \beta], o, (S, S_0), \{n\}, \zeta \cup \{n \overset{m}{\mapsto} \alpha, o \overset{n}{\mapsto} \beta\})}$$

(Choice)

$$\frac{}{(P \sqcap_\alpha Q, G, m, (S, S_0), \Delta, \zeta) \xrightarrow{\tau} (P', G[n \overset{m}{\mapsto} \alpha], n, (S', S'_0), \emptyset, \zeta \cup \{n \overset{m}{\mapsto} \alpha\})}$$

$$(P', (S', S'_0)) \;=\; \texttt{SelectBranch}(P \sqcap_\alpha Q, (S, S_0))$$

(STOP)

$$\frac{}{(STOP_\alpha, G, m, (S, S_0), \Delta, \zeta) \xrightarrow{\tau} (\bot, G[n \overset{m}{\mapsto} \alpha], n, (S, S_0), \emptyset, \zeta \cup \{n \overset{m}{\mapsto} \alpha\})}$$

Figure 3: An instrumented operational semantics that generates the CSCFG

(Process Call) The called process N is unfolded, node n (a fresh reference) is added to the graphs G and ζ with specification position α and parent m. In the new state, n represents the current reference. The new expression in the control is P', computed with function `LoopCheck` which is used to prevent infinite unfolding and is defined below. No event can synchronize in this rule, thus Δ is empty.

$$\texttt{LoopCheck}(N, n, G, \zeta) = \begin{cases} (\circlearrowleft_s(rhs(N)), G[n \rightsquigarrow s], \zeta \cup \{n \rightsquigarrow s\}) & \text{if } \exists s.s \overset{t}{\mapsto} N \in G \\ & \wedge s \in Path(0, n) \\ (rhs(N), G, \zeta) & \text{otherwise} \end{cases}$$

Function `LoopCheck` checks whether the process call in the control has not been already executed (if so, we are in a loop). When a loop is detected, a loop edge between nodes n and s is added to the graph G and to ζ; and the right-hand side of the called process is labelled with a special symbol $\circlearrowleft_s$. This label is later used by rule (Synchronized Parallelism 4) to decide whether the process must be stopped. The loop symbol $\circlearrowleft$ is labelled with the position s of the process call of the loop. This is used to know what is the reference of the process' node if it is unfolded again.

(Prefixing) This rule adds nodes n (the prefix) and o (the prefixing operator) to the graphs G and ζ. In the new state, o becomes the current reference. The new control is P. The set Δ is $\{n\}$ to indicate that event a has occurred and it must be synchronized when required by (Synchronized Parallelism 3).

(Choice) The only sources of non-determinism are choice operators (different

Figure 3: An instrumented operational semantics that generates the CSCFG (cont.)

branches can be selected for execution) and parallel operators (different order of branches can be selected for execution). Therefore, every time the semantics executes a choice or a parallelism, they are made deterministic thanks to the information in the stack S. Both internal and external can be treated with

a single rule because the CSCFG associated to a specification with external choices is identical to the CSCFG associated to the specification with the external choices replaced by internal choices. This rule adds node n to the graphs which is labelled with the specification position α and has parent m. In the new state, n becomes the current reference. No event can synchronize in this rule, thus Δ is empty.

Function `SelectBranch` is used to produce the new control P' and the new tuple of stacks (S', S'_0), by selecting a branch with the information of the stack. Note that, for simplicity, the lists constructor ":" has been overloaded, and it is also used to build lists of the form $(A : a)$ where A is a list and a is the last element:

$$\texttt{SelectBranch}(P \sqcap_\alpha Q, (S, S_0)) = \begin{cases} (P, (S', (\mathsf{C1}, \{\mathsf{C2}\}) : S_0)) & \text{if } S = S' : (\mathsf{C1}, \{\mathsf{C2}\}) \\ (Q, (S', (\mathsf{C2}, \emptyset) : S_0)) & \text{if } S = S' : (\mathsf{C2}, \emptyset) \\ (P, (\emptyset, (\mathsf{C1}, \{\mathsf{C2}\}) : S_0)) & \text{otherwise} \end{cases}$$

If the last element of the stack S indicates that the first branch of the choice ($\mathsf{C1}$) must be selected, then P is the new control. If the second branch must be selected ($\mathsf{C2}$), the new control is Q. In any other case the stack is empty, and thus this is the first time that this choice is evaluated. Then, we select the first branch (P is the new control) and we add $(\mathsf{C1}, \{\mathsf{C2}\})$ to the stack S_0 indicating that $\mathsf{C1}$ has been fired, and the remaining option is $\mathsf{C2}$.

For instance, when the CSCFG of Fig. 2(b) is being constructed and we reach the choice operator (i.e., $(\texttt{MAIN}, 2)$), then the left branch of the choice is evaluated and $(\mathsf{C1}, \{\mathsf{C2}\})$ is added to the stack to indicate that the left branch has been evaluated. The second time it is evaluated, the stack is updated to $(\mathsf{C2}, \emptyset)$ and the right branch is evaluated. Therefore, the selection of branches is predetermined by the stack, thus, Algorithm 1 can decide what branches are evaluated by conveniently handling the information of the stack.

(Synchronized Parallelism 1 and 2) The stack determines what rule to use when a parallelism operator is in the control. If the last element in the stack is SP1, then (Synchronized Parallelism 1) is used. If it is SP2, (Synchronized Parallelism 2) is used.

In a synchronized parallelism composition, both parallel processes can be intertwiningly executed until a synchronized event is found. Therefore, nodes for both processes can be added interwoven to the graph. Hence, the semantics needs to know in every state the references to be used in both branches. This is done by labelling each parallelism operator with a tuple of the form $(\alpha, n_1, n_2, \Upsilon)$ where α is the specification position of the parallelism operator; n_1 and n_2 are respectively the references of the last node introduced in the left and right branches of the parallelism, and they are initialised to $\bullet$; and Υ is a node reference used to decide when to unfold a process call (in order to avoid infinite loops), also initialised to $\bullet$. The sets Δ' and ζ'' are passed down unchanged so that another rule can use them if necessary. In the case that ζ is equal to ζ'', meaning that nothing has change in this derivation, this rule detects that the parallelism is in a loop; and thus, in the new control the parallelism operator is labelled with $\circlearrowleft$ and all the other loop labels are removed from it (this is done by a trivial function `Unloop`).

These rules develop the branches of the parallelism until they are finished or until they must synchronize. They use function `InitBranch` to introduce the parallelism into the graph and into ζ the first time it is executed and only if

it has not been introduced in a previous computation. For instance, consider a state where a parallelism operator is labelled with $((\texttt{MAIN}, \Lambda), \bullet, \bullet, \bullet)$. Therefore, it is evaluated for the first time, and thus, when, e.g., rule (Synchronized Parallelism 1) is applied, a node $1 \overset{0}{\mapsto} (\texttt{MAIN}, \Lambda)$, which refers to the parallelism operator, is added to G and the parallelism operator is relabelled to $((\texttt{MAIN}, \Lambda), \texttt{x}, \bullet, \bullet)$ where x is the new reference associated with the left branch. After executing function `InitBranch`, we get a new graph and a new reference. Its definition is the following:

$$\texttt{InitBranch}(G, \zeta, n, m, \alpha) = \begin{cases} (G[o \overset{m}{\mapsto} \alpha], \zeta \cup \{o \overset{m}{\mapsto} \alpha\}, o) & \text{if } n = \bullet \\ (G, \zeta, n) & \text{otherwise} \end{cases}$$

(Synchronized Parallelism 3) It is applied when the last element in the stack is SP3. It is used to synchronize the parallel processes. In this rule, Υ is replaced by $\bullet$, meaning that a synchronization edge has been drawn and the loops could be unfolded again if it is needed. The set *sync* of all the events that have been executed in this step must be synchronized. Therefore, all the events occurred in the subderivations of $P1$ (Δ_1) and $P2$ (Δ_2) are mutually synchronized and added to both G'' and ζ'.

(Synchronized Parallelism 4) This rule is applied when the last element in the stack is SP4. It is used when none of the parallel processes can proceed (because they already finished, deadlocked or were labelled with $\circlearrowleft$). When a process is labelled as a loop with $\circlearrowleft$, it can be unlabelled to unfold it once[3] in order to allow the other processes to continue. This happens when the looped process is in parallel with other process and the later is waiting to synchronize with the former. In order to perform the synchronization, both processes must continue, thus the loop is unlabelled. Hence, the system must stop only when both parallel processes are marked as a loop. This task is done by function `LoopControl`. It decides if the branches of the parallelism should be further unfolded or they should be stopped (e.g., due to a deadlock or an infinite loop):

$$\texttt{LoopControl}(P \underset{X}{\|}_{(\alpha,p,q,\Upsilon)} Q, m) =$$

$$\begin{cases} \circlearrowleft_m(P'_{\circlearrowleft} \underset{X}{\|}_{(\alpha,p_{\circlearrowleft},q_{\circlearrowleft},\bullet)} Q'_{\circlearrowleft}) & \text{if } P' = \circlearrowleft_{p_{\circlearrowleft}}(P'_{\circlearrowleft}) \wedge Q' = \circlearrowleft_{q_{\circlearrowleft}}(Q'_{\circlearrowleft}) \\ \circlearrowleft_m(P'_{\circlearrowleft} \underset{X}{\|}_{(\alpha,p_{\circlearrowleft},q',\bullet)} \bot) & \text{if } P' = \circlearrowleft_{p_{\circlearrowleft}}(P'_{\circlearrowleft}) \ \wedge (Q' = \bot \vee (\Upsilon = p_{\circlearrowleft} \wedge Q' \neq \circlearrowleft\_(\_))) \\ P'_{\circlearrowleft} \underset{X}{\|}_{(\alpha,p_{\circlearrowleft},q',p_{\circlearrowleft})} Q' & \text{if } P' = \circlearrowleft_{p_{\circlearrowleft}}(P'_{\circlearrowleft}) \ \wedge Q' \neq \bot \wedge \ \Upsilon \neq p_{\circlearrowleft} \wedge Q' \neq \circlearrowleft\_(\_) \\ \bot & \text{otherwise} \end{cases}$$

where $(P', p', Q', q') \in \{(P, p, Q, q), (Q, q, P, p)\}$.

When one of the branches has been labelled as a loop, there are three options: (i) The other branch is also a loop. In this case, the whole parallelism is marked as a loop labelled with its parent, and Υ is put to $\bullet$. (ii) Either it is a loop that has been unfolded without drawing any synchronization (this is known because Υ is equal to the parent of the loop), or the other branch already terminated (i.e., it is $\bot$). In this case, the parallelism is also marked as a loop, and the other branch is put to $\bot$ (this means that this process has been deadlocked). Also

[3] Only once because it will be labelled again by rule (Process Call) when the loop is repeated. In [10], we present an example with loops where this situation happens.

here, Υ is put to $\bullet$. (iii) If we are not in a loop, then we allow the parallelism to proceed by unlabelling the looped branch. When none of the branches has been labelled as a loop, $\bot$ is returned representing that this is a deadlock, and thus, stopping further computations.

(Synchronized Parallelism 5) This rule is used when the stack is empty. It basically analyses the control and decides what are the applicable rules of the semantics. This is done with function `AppRules` which returns the set of rules R that can be applied to a synchronized parallelism $P \underset{X}{\|} Q$:

$$\texttt{AppRules}(P \underset{X}{\|} Q) = \begin{cases} \{\mathsf{SP1}\} & \text{if } \tau \in \texttt{FstEvs}(P) \\ \{\mathsf{SP2}\} & \text{if } \tau \notin \texttt{FstEvs}(P) \wedge \tau \in \texttt{FstEvs}(Q) \\ R & \text{if } \tau \notin \texttt{FstEvs}(P) \wedge \tau \notin \texttt{FstEvs}(Q) \ \wedge \ R \neq \emptyset \\ \{\mathsf{SP4}\} & \text{otherwise} \end{cases}$$

where

$$\begin{cases} \mathsf{SP1} \in R & \text{if } \exists e \in \texttt{FstEvs}(P) \wedge e \notin X \\ \mathsf{SP2} \in R & \text{if } \exists e \in \texttt{FstEvs}(Q) \wedge e \notin X \\ \mathsf{SP3} \in R & \text{if } \exists e \in \texttt{FstEvs}(P) \ \wedge \exists e \in \texttt{FstEvs}(Q) \wedge e \in X \end{cases}$$

Essentially, `AppRules` decides what rules are applicable depending on the events that could happen in the next step. These events can be inferred by using function `FstEvs`. In particular, given a process P, function `FstEvs` returns the set of events that can fire a rule in the semantics using P as the control. Therefore, rule (Synchronized Parallelism 5) prepares the stack allowing the semantics to proceed with the correct rule.

$\texttt{FstEvs}(P) =$

$$\begin{cases} \{a\} & \text{if } \ P = a \rightarrow Q \\ \emptyset & \text{if } \ P = \circlearrowleft Q \ \vee \ P = \bot \\ \{\tau\} & \text{if } \ P = M \vee P = \texttt{STOP} \vee P = Q \sqcap R \vee P = (\bot \underset{X}{\|} \bot) \\ & \quad \vee \ P = (\circlearrowleft Q \underset{X}{\|} \circlearrowleft R) \ \vee \ P = (\circlearrowleft Q \underset{X}{\|} \bot) \ \vee \ P = (\bot \underset{X}{\|} \circlearrowleft R) \\ & \quad \vee (P = (\circlearrowleft Q \underset{X}{\|} R) \wedge \texttt{FstEvs}(R) \subseteq X) \vee (P = (Q \underset{X}{\|} \circlearrowleft R) \wedge \texttt{FstEvs}(Q) \subseteq X) \\ & \quad \vee (P = Q \underset{X}{\|} R \wedge \texttt{FstEvs}(Q) \subseteq X \wedge \texttt{FstEvs}(R) \subseteq X \wedge \bigcap_{M \in \{Q,R\}} \texttt{FstEvs}(M) = \emptyset) \\ E & \text{otherwise, where } P = Q \underset{X}{\|} R \wedge E = (\texttt{FstEvs}(Q) \cup \texttt{FstEvs}(R)) \setminus \\ & \qquad\qquad (X \cap (\texttt{FstEvs}(Q) \setminus \texttt{FstEvs}(R) \cup \texttt{FstEvs}(R) \setminus \texttt{FstEvs}(Q))) \end{cases}$$

(STOP) Whenever this rule is applied, the subcomputation finishes because $\bot$ is put in the control, and this special constructor has no associated rule. A node with the `STOP` position is added to the graph.

We illustrate this semantics with a simple example.

Example 3. *Consider again the specification in Example 1. Due to the choice operator, in this specification two different events can occur, namely* `b` *and* `a`*. Therefore, Algorithm 1 performs two iterations (one for each computation) to generate the final CSCFG. Figure 2(b) shows the CSCFG generated where white nodes were produced in the first iteration; and grey nodes were produced in the second iteration. For the interested reader, in [10] all computation steps executed by Algorithm 1 to obtain the CSCFG associated with the specification in Example 1 are explained in detail.*

5. Correctness

In this section we state the correctness of the proposed algorithm by showing that (i) the graph produced by the algorithm for a CSP specification $\mathcal{S}$ is the CSCFG of $\mathcal{S}$; and (ii) the algorithm terminates, even if non-terminating computations exist for the specification $\mathcal{S}$.

Theorem 4 (Correctness). *Let $\mathcal{S}$ be a CSP specification and G the graph produced for $\mathcal{S}$ by Algorithm 1. Then, G is the CSCFG associated with $\mathcal{S}$.*

This theorem can be proved by showing first that each step performed with the standard semantics has an associated step in the instrumented semantics; and that the specification position of the expression in the control is added to the CSCFG as a new node which is properly inserted into the CSCFG. This can be proved by induction on the length of a derivation in the standard semantics. Then, it must be proved that the algorithm performs all possible computations. This can be done by showing that every non-deterministic step of the semantics is recorded in the stack with all possible rules that can be applied; and the algorithm traverses the stack until all possibilities have been evaluated. The interesting case of the proof happens when the computation is infinite. In this case, the context of a process call must be repeated because the number of process calls is finite by definition. Therefore, in this case the proof must show that functions `LoopCheck` and `LoopControl` correctly finish the computation. The proof of this theorem can be found in [10].

Theorem 5 (Termination). *Let $\mathcal{S}$ be a CSP specification. Then, the execution of Algorithm 1 with $\mathcal{S}$ terminates.*

The proof of this theorem must ensure that all derivations of the instrumented semantics are finite, and that the number of derivations fired by the algorithm is also finite. This can be proved by showing that the stacks never grow infinitely, and they will eventually become empty after all computations have been explored. The proof of this theorem can be found in [10].

6. Conclusions

This work introduces an algorithm to build the CSCFG associated with a CSP specification. The algorithm uses an instrumentation of the standard CSP's operational semantics to explore all possible computations of a specification. The semantics is deterministic because the rule applied in every step is predetermined by the initial state and the information in the stack. Therefore, the algorithm can execute the semantics several times to iteratively explore all computations and hence, generate the whole CSCFG. The CSCFG is generated even for non-terminating specifications due to the use of a loop detection mechanism controlled by the semantics. This semantics is an interesting result because it can serve as a reference mark to prove properties such as completeness of static analyses based on the CSCFG. The way in which the semantics has been instrumented can be used for other similar purposes with slight modifications. For instance, the same design could be used to generate other graph representations of a computation such as Petri nets [11].

On the practical side, we have implemented a tool called *SOC* [8] which is able to automatically generate the CSCFG of a CSP specification. The CSCFG is later used for debugging and program simplification. *SOC* has been integrated into the most extended CSP animator and model-checker ProB [2, 6], that shows the maturity and usefulness of this tool and of CSCFGs. The last release of *SOC* implements the algorithm described in this paper. However, in the implementation the algorithm is much more complex because it contains some improvements that significantly speed up the CSCFG construction. The most important improvement is to avoid repeated computations. This is done by: (i) state memorization: once a state already explored is reached the algorithm stops this computation and starts with another one; and (ii) skipping already performed computations: computations do not start from `MAIN`, they start from the next non-deterministic state in the execution (this is provided by the information of the stack). The implementation, source code and several examples are publicly available at: `http://users.dsic.upv.es/~jsilva/soc/`

References

[1] Brassel, B., Hanus, M., Huch, F., Vidal, G.: A Semantics for Tracing Declarative Multi-paradigm Programs. In: Moggi, E., Warren, D.S. (eds.) 6th ACM SIGPLAN Int'l Conf. on Principles and Practice of Declarative Programming (PPDP'04), pp. 179–190. ACM, New York, NY, USA (2004)

[2] Butler, M., Leuschel, M.: Combining CSP and B for Specification and Property Verification. In: Fitzgerald, J., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 221–236. Springer, Heildeberg (2005)

[3] Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Upper Saddle River, NJ, USA (1985)

[4] Kavi, K.M., Sheldon, F.T., Shirazi, B., Hurson, A.R.: Reliability Analysis of CSP Specifications using Petri Nets and Markov Processes. In: 28th Annual Hawaii Int'l Conf. on System Sciences (HICSS'95), vol. 2 (Software Technology), pp. 516–524. IEEE Computer Society, Washington, DC, USA (1995)

[5] Ladkin, P., Simons, B.: Static Deadlock Analysis for CSP-Type Communications. Responsive Computer Systems (Chapter 5), Kluwer Academic Publishers (1995)

[6] Leuschel, M., Butler, M.: ProB: an Automated Analysis Toolset for the B Method. Journal of Software Tools for Technology Transfer. 10(2), 185–203 (2008)

[7] Leuschel, M., Llorens, M., Oliver, J., Silva, J., Tamarit, S.: Static Slicing of CSP Specifications. In: Hanus, M. (ed.) 18th Int'l Symp. on Logic-Based Program Synthesis and Transformation (LOPSTR'08), pp. 141–150. Technical report, DSIC-II/09/08, Universidad Politécnica de Valencia (July 2008)

[8] Leuschel, M., Llorens, M., Oliver, J., Silva, J., Tamarit, S.: SOC: a Slicer for CSP Specifications. In: Puebla, G., Vidal, G. (eds.) 2009 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM'09), pp. 165–168. ACM, New York, NY, USA (2009)

[9] Leuschel, M., Llorens, M., Oliver, J., Silva, J., Tamarit, S.: The MEB and CEB Static Analysis for CSP Specifications. In: Hanus, M. (ed.) LOPSTR 2008, Revised Selected Papers. LNCS, vol. 5438, pp. 103–118. Springer, Heildeberg (2009)

[10] Llorens, M., Oliver, J., Silva, J., Tamarit, S.: A Semantics to Generate the Context-sensitive Synchronized Control-Flow Graph (extended). Technical report DSIC, Universidad Politécnica de Valencia. Accessible via `http://www.dsic.upv.es/~jsilva`, Valencia, Spain, June 2010.

[11] Llorens, M., Oliver, J., Silva, J., Tamarit, S.: Transforming Communicating Sequential Processes to Petri Nets. In: Topping, B.H.V., Adam, J.M., Pallarés, F.J., Bru, R., Romero, M.L. (eds.) Seventh Int'l Conf. on Engineering Computational Technology (ICECT'10). Civil-Comp Press, Stirlingshire, Scotland (to appear 2010)

[12] Roscoe, A.W., Gardiner, P.H.B., Goldsmith, M., Hulance, J.R., Jackson, D.M., Scattergood, J.B.: Hierarchical Compression for Model-Checking CSP or How to Check 10^{20} Dining Philosophers for Deadlock. In: Brinksma, E., Cleaveland, R., Larsen, K.G., Margaria, T., Steffen, B. (eds.) TACAS 1995. LNCS, vol. 1019, pp. 133–152. Springer, London (1995)

[13] Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice Hall, Upper Saddle River, NJ, USA (2005)

A Tracking Semantics for CSP[☆]

Marisa Llorens[a], Javier Oliver[a], Josep Silva[a], Salvador Tamarit[a]

[a] *Universitat Politècnica de València, Camino de Vera S/N, E-46022 Valencia, Spain*

Abstract

CSP is a powerful language for specifying complex concurrent systems. Due to the non-deterministic execution order of processes and to the restrictions imposed on this order by synchronizations, many analyses such as deadlock analysis, reliability analysis, and program slicing try to predict properties of the specification which can guarantee the quality of the final system. These analyses often rely on the use of CSP's traces. In this work, we introduce the theoretical basis for tracking concurrent and explicitly synchronized computations in process algebras such as CSP. Tracking computations is a difficult task due to the subtleties of the underlying operational semantics which combines concurrency, non-determinism and non-termination. We define an instrumented operational semantics that generates as a side-effect an appropriate data structure (a track) which can be used to track computations at an adequate level of abstraction. Given an execution, its track is more informative than its trace since the former not only contains a lot of information about original program structures but also explicitly relates the sequence of events with the parts of the specification that caused these events. Formal definition of a tracking semantics improves the understanding of the tracking process, but also, it allows to formally prove the correctness of the computed tracks.

1. Introduction

One of the most important techniques for program understanding and debugging is tracing [3]. A trace gives the user access to otherwise invisible information about a computation. In the context of concurrent languages, computations are particularly complex due to the non-deterministic execution order of processes and to the restrictions imposed on this order by synchronizations; and thus, a tracer is a powerful tool to explore, understand and debug concurrent computations.

[☆]This work has been partially supported by the Spanish *Ministerio de Ciencia e Innovación* under grant TIN2008-06622-C03-02, by the *Generalitat Valenciana* under grant ACOMP/2009/017, and by the *Universidad Politécnica de Valencia* (Programs PAID-05-08 and PAID-06-08). Salvador Tamarit was partially supported by the Spanish MICINN under FPI grant BES-2009-015019.

Email addresses: mllorens@dsic.upv.es (Marisa Llorens), fjoliver@dsic.upv.es (Javier Oliver), jsilva@dsic.upv.es (Josep Silva), stamarit@dsic.upv.es (Salvador Tamarit)

One of the most widespread concurrent specification languages is the *Communicating Sequential Processes* (CSP) [7, 16] whose operational semantics allows us the combination of parallel, non-deterministic and non-terminating processes. The study and transformation of CSP specifications often uses different analyses such as deadlock analysis [10], reliability analysis [8] and program slicing [18] which are based on a data structure able to represent computations.

In CSP a trace is a sequence of events. Concretely, the operational semantics of CSP is an event-based semantics in which the occurrence of events fires the rules of the semantics. Hence, the final trace of the computation is the sequence of events occurred (see Chapter 8 of [16] for a detailed study of this kind of traces). In this work we introduce an essentially different notion of trace [3] called track. In our setting, a track is a data structure which represents the sequence of expressions that have been evaluated during the computation, and moreover, this data structure is labelled with the location of these expressions in the specification. Therefore, a CSP track is much more informative than a CSP trace since the former not only contains a lot of information about original program structures but also explicitly relates the sequence of events with the parts of the specification that caused these events.

Example 1. *Consider the following CSP specification*[1]*:*

$$\underline{\mathit{MAIN}} = \underline{\mathit{CASINO} \parallel \mathit{GAMBLING}}$$

$$\underline{\texttt{CASINO}} = \underline{(\texttt{PLAYER} \,|||\, \texttt{ROULETTE})} \underset{\{\texttt{betred},\texttt{red},\texttt{black},\texttt{prize}\}}{\parallel} \underline{\texttt{CROUPIER}}$$

$$\underline{\texttt{PLAYER}} = \underline{\texttt{betred} \rightarrow} \; (\underline{\texttt{prize} \rightarrow \; \texttt{STOP}} \; \Box \; \texttt{noprize} \rightarrow \texttt{STOP})$$

$$\underline{\texttt{ROULETTE}} = \texttt{red} \rightarrow \texttt{STOP} \; \underline{\Box \; \texttt{black} \rightarrow \; \texttt{STOP}}$$

$$\begin{aligned} \underline{\texttt{CROUPIER}} = \; & (\texttt{betred} \rightarrow \texttt{red} \rightarrow \texttt{prize} \rightarrow \texttt{STOP}) \\ & \underline{\Box \; (\texttt{betred} \rightarrow \texttt{black} \rightarrow \texttt{prize} \rightarrow \; \texttt{STOP})} \\ & \Box \; (\texttt{betblack} \rightarrow \texttt{black} \rightarrow \texttt{prize} \rightarrow \texttt{STOP}) \\ & \Box \; (\texttt{betblack} \rightarrow \texttt{red} \rightarrow \texttt{getmoney} \rightarrow \texttt{STOP}) \end{aligned}$$

$\texttt{GAMBLING} =$ *Complex Composite Processes*

This specification models several gambling activities running in parallel and modeled by process GAMBLING*. One of the games is the casino. A* CASINO *is modeled as the interaction of three parallel processes, namely a* PLAYER*, a* ROULETTE*, and a* CROUPIER*. The player bets for red, and she can win a prize or not. The roulette simply takes a color (either red or black); and the croupier checks the bet and the color of the roulette in order to give a prize to the player or just get the bet money.*

This specification contains an error, because it allows the trace of events $t = \langle \texttt{betred}, \texttt{black}, \texttt{prize} \rangle$ *where the player bets for red and she wins a prize even though the roulette takes black.*

[1] We refer those readers non familiarized with CSP syntax to Section 2 where we provide a brief introduction to CSP.

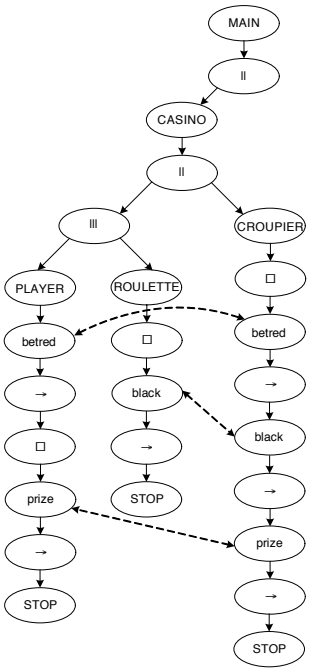


Figure 1: Track of the program in Example 1

Now assume that we execute the specification and discover the error after executing trace t. A track can be very useful to understand why the error was caused, and what part of the specification was involved in the wrong execution. For instance, if we look at the track of Fig. 1, we can easily see that the three processes run in parallel, and that the prize is given because there is a synchronization (dashed edges represent synchronizations) between `CROUPIER` *and* `PLAYER` *that should never happen. Observe that the track is intuitive enough as to be a powerful program comprehension tool that provides much more information than the trace.*

Moreover, observe that the track contains explicit information about the specification expressions that were involved in the execution. Therefore, it can be used for program slicing (see [17] for an explanation of the technique and [14] for an adaptation of program slicing to CSP). In particular, in this example, we can use the track to extract the part of the program that was involved in the execution—note that this is the only part that could cause the error—. This part has been underscored in the example. With a quick look, one can see that the underscored part of process `CROUPIER` *produced the wrong behavior. Event* `prize` *should be replaced by* `getmoney`.

Another interesting application of tracks is related to component extraction and reuse. If we are interested in a particular trace, and we want to extract the part of the specification that models this trace to be used in another model, we can simply produce a slice, and slightly augment the code to make it syntactically correct (see [14] for an example and an explanation of this transformation). In our example, even though the system is very big due to the process `GAMBLING`*, the track is able to extract the only information related to the trace.*

We have implemented a tool [13] able to produce tracks and to automatically color parts of the code related to some point in the specification. This tool is integrated in the last version of ProB [11, 12] which is the most extended IDE for CSP.

In languages such as Haskell, the tracks (see, e.g., [3, 4, 5, 1]) are the basis of many analysis methods and tools. However, computing CSP tracks is a complex task due to the non-deterministic execution of processes, due to deadlocks, due to non-terminating processes and mainly due to synchronizations. This is probably the reason why no correctness result exists which formally relates the track of a specification to its execution. This semantics is needed because it would allow us to prove important properties (such as correctness and completeness) of the techniques and tools based on tracking.

To the best of our knowledge, there is only one attempt to define and build tracks for CSP [2]. Their notion of track is based on the standard *program dependence graph* [6]; therefore it is useful for program slicing but it is insufficient for other analyses that need a *context-sensitive graph* [9] (i.e., each different process call has a different representation). Moreover, their notion of track does not include synchronizations. Our tracks are able to represent synchronizations, and they are context-sensitive.

The main contributions of this work are the formal definition of tracks, the definition of the first tracking semantics for CSP and the proof that the trace of a computation can be extracted from the track of this computation. Concretely, we instrument the standard operational semantics of CSP in such a way that the execution of the semantics produces as a side-effect the track of the computation. It should be clear that the track of an infinite computation is also infinite. However, we design the semantics in such a way that the track is produced incrementally step by step. Therefore, if the execution is stopped (e.g., by the user because it is non-terminating or because a limit in the size of the track was specified), then the semantics produces the track of the computation performed so far. This semantics can serve as a theoretical foundation for tracking CSP computations because it formally relates the computations of the standard semantics with the tracks of these computations.

The rest of the paper has been organized as follows. Firstly, in Section 2 we recall the syntax and semantics of CSP. In Section 3 we define the concept of track for CSP. Then, in Section 4, we instrument the CSP semantics in such a way that its execution produces as a side-effect the track associated with the performed computation. In Section 5, we present the main results of the paper proving that the instrumented semantics presented is a conservative extension of the standard semantics, its computed tracks are correct and the corresponding trace can be extracted from the track. Finally, Section 6 concludes.

2. The syntax and semantics of CSP

In order to make the paper self-contained, we recall in this section the syntax and semantics of CSP.

Figure 2 summarizes the syntax constructions used in CSP specifications. A *specification* is viewed as a finite set of process definitions. The left-hand side of each definition is the name of a process, which is defined in the right-hand side

(abbrev. *rhs*) by means of an expression that can be a call to another process or a combination of the following operators:

Prefixing It specifies that event a must happen before process P.

Internal choice The system chooses non-deterministically to execute one of the two processes P or Q.

External choice It is identical to internal choice but the choice comes from outside the system (e.g., the user).

Sequential composition It specifies a sequence of two processes. When the first (successfully) finishes, the second starts.

Synchronized parallelism Both processes are executed in parallel with a set X of synchronized events. In absence of synchronizations both processes can execute in any order. Whenever a synchronized event $a \in X$ happens in one of the processes it must also happen in the other at the same time. Whenever the set of synchronized events is not specified, it is assumed that processes are synchronized in all common events. A particular case of parallel execution is *interleaving* where no synchronizations exist (i.e., $X = \emptyset$).

Skip It successfully finishes the current process. It allows us to continue the next sequential process.

Stop Synonymous of deadlock: It finishes the current process and it does not allow the next sequential process to continue.

S	::=	$\{D_1, \ldots, D_m\}$	(Entire specification)	*Domains* $M, N \ldots \in \mathcal{N}$ (Process names) $P, Q \ldots \in \mathcal{P}$ (Processes) $a, b \ldots \in \Sigma$ (Events)
D	::=	$N = P$	(Process definition)	
P	::=	M	(Process call)	
	|	$a \rightarrow P$	(Prefixing)	
	|	$P \sqcap Q$	(Internal choice)	
	|	$P \,\Box\, Q$	(External choice)	
	|	$P \,;\, Q$	(Sequential composition)	
	|	$P \underset{X}{\|\|} Q$	(Synchronized parallelism)	where $X \subseteq \Sigma$
	|	$SKIP$	(Skip)	
	|	$STOP$	(Stop)	

Figure 2: Syntax of CSP specifications

We now recall the standard operational semantics of CSP as defined by Roscoe [16]. It is presented in Fig. 3 as a logical inference system. A *state* of the semantics is a process to be evaluated called the *control*. The system starts with an initial state, and the rules of the semantics are used to infer how this state evolves. When no rules can be applied to the current state, the computation finishes. The rules of the semantics change the states of the computation due to the occurrence of events. The set of possible events is $\Sigma \cup \{\tau, \checkmark\}$. Events in $\Sigma = \{a, b, c \ldots\}$ are visible from the external environment, and can only happen with its co-operation (e.g., actions of the user). The special event τ cannot

be observed from outside the system and it is an internal event that happens automatically as defined by the semantics. ✓ is a special event representing the successful termination of a process. We use the special symbol Ω to denote any process that successfully terminated.

In order to perform computations, we construct an initial state (e.g., `MAIN`) and (non-deterministically) apply the rules of Fig. 3. The intuitive meaning of each rule is the following:

(Process Call) The call is unfolded and the right-hand side of process named N is added to the control.

(Prefixing) When event a occurs, process P is added to the control.

(SKIP) After `SKIP`, the only possible event is ✓, which denotes the successful termination of the (sub)computation with the special symbol Ω. There is no rule for Ω (neither for `STOP`), hence, this (sub)computation has finished.

(Internal Choice 1 and 2) The system, with the occurrence of the internal event τ, (non-deterministically) selects one of the two processes P or Q which is added to the control.

(External Choice 1, 2, 3 and 4) The occurrence of τ develops one of the branches. The occurrence of an event $a \neq \tau$ is used to select one of the two processes P or Q and the control changes according to the event.

(Sequential Composition 1) In $P;Q$, P can evolve to P' with any event except ✓. Hence, the control becomes $P';Q$.

(Sequential Composition 2) When P successfully finishes (with event ✓), Q starts. Note that ✓ is hidden from outside the whole process becoming τ.

(Synchronized Parallelism 1 and 2) When event $a \notin X$ or events τ or ✓ happen, one of the two processes P or Q evolves accordingly, but only a is visible from outside the parallelism operator.

(Synchronized Parallelism 3) When event $a \in X$ happens, it is required that both processes synchronize, P and Q are executed at the same time and the control becomes $P' \underset{X}{||} Q'$.

(Synchronized Parallelism 4) When both processes have successfully terminated the control becomes Ω, performing the event ✓.

(Process Call)

$$\frac{}{N \xrightarrow{\tau} rhs(N)}$$

(Prefixing)

$$\frac{}{(a \to P) \xrightarrow{a} P}$$

(SKIP)

$$\frac{}{\texttt{SKIP} \xrightarrow{\checkmark} \Omega}$$

(Internal Choice 1)

$$\frac{}{(P \sqcap Q) \xrightarrow{\tau} P}$$

(Internal Choice 2)

$$\frac{}{(P \sqcap Q) \xrightarrow{\tau} Q}$$

(External Choice 1)

$$\frac{P \xrightarrow{\tau} P'}{(P \,\Box\, Q) \xrightarrow{\tau} (P' \,\Box\, Q)}$$

(External Choice 2)

$$\frac{Q \xrightarrow{\tau} Q'}{(P \,\Box\, Q) \xrightarrow{\tau} (P \,\Box\, Q')}$$

(External Choice 3)

$$\frac{P \xrightarrow{e} P'}{(P \,\Box\, Q) \xrightarrow{e} P'}$$

(External Choice 4)

$$\frac{Q \xrightarrow{e} Q'}{(P \,\Box\, Q) \xrightarrow{e} Q'} \quad e \in \Sigma \cup \{\checkmark\}$$

(Sequential Composition 1)

$$\frac{P \xrightarrow{e} P'}{(P;Q) \xrightarrow{e} (P';Q)} \quad e \in \Sigma \cup \{\tau\}$$

(Sequential Composition 2)

$$\frac{P \xrightarrow{\checkmark} \Omega}{(P;Q) \xrightarrow{\tau} Q}$$

(Synchronized Parallelism 1)

$$\frac{P \xrightarrow{e'} P'}{(P \underset{X}{||} Q) \xrightarrow{e} (P' \underset{X}{||} Q)}$$

(Synchronized Parallelism 2)

$$\frac{Q \xrightarrow{e'} Q'}{(P \underset{X}{||} Q) \xrightarrow{e} (P \underset{X}{||} Q')} \quad (e = e' = a \ \wedge\ a \notin X) \vee (e = \tau \ \wedge\ e' \in \{\tau, \checkmark\})$$

(Synchronized Parallelism 3)

$$\frac{P \xrightarrow{a} P' \qquad Q \xrightarrow{a} Q'}{(P \underset{X}{||} Q) \xrightarrow{a} (P' \underset{X}{||} Q')} \quad a \in X$$

(Synchronized Parallelism 4)

$$\frac{}{(\Omega \underset{X}{||} \Omega) \xrightarrow{\checkmark} \Omega}$$

Figure 3: CSP's operational semantics

We illustrate the semantics with the following example.

Example 2. *Consider the next CSP specification:*

$$\texttt{MAIN} \;=\; (\texttt{a} \to \texttt{STOP}) \underset{\{\texttt{a}\}}{\|} (\texttt{P} \;\Box\; (\texttt{a} \to \texttt{STOP}))$$

$$\texttt{P} \;=\; \texttt{b} \to \texttt{SKIP}$$

If we use `MAIN` *as the initial state to execute the semantics, we get the computation shown in Fig. 4 where the final state is* $((\texttt{a} \to \texttt{STOP}) \underset{\{\texttt{a}\}}{\|} \Omega)$*. This computation corresponds to the execution of the left branch of the choice (i.e.,* `P`*) and thus only event* `b` *occurs. Each rewriting step is labelled with the applied rule, and the example should be read top-down.*

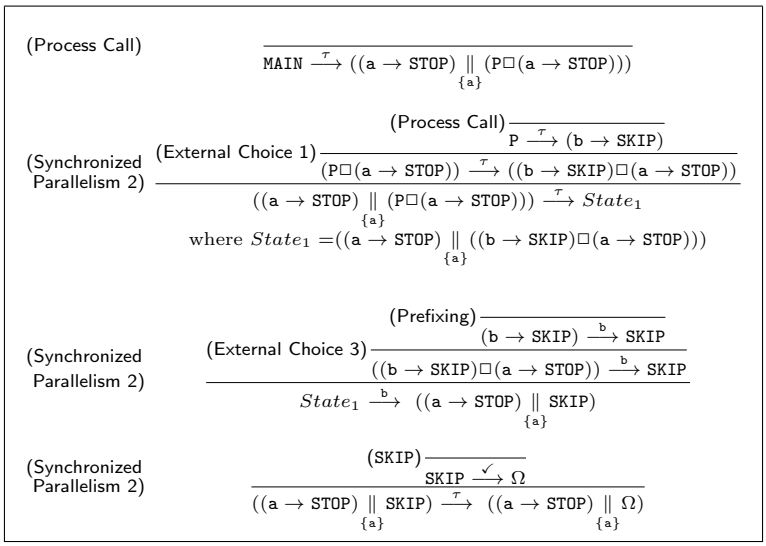


Figure 4: A computation with the operational semantics in Fig. 3

3. Tracking computations

In this section we define the notion of track. Firstly, we introduce some notation that will be used throughout the paper.

A track is formed by the sequence of expressions that are evaluated during an execution. These expressions are conveniently connected to form a graph. However, several program analysis techniques such as program slicing handle the locations of program expressions, and thus, this notion of track is insufficient for them. Therefore, we want our tracks to also store the location of each literal (i.e., events, operators and process names) in the specification so that the track can be used to know what portions of the source code have been executed and in what order. The inclusion of source positions in the track implies an additional level of complexity in the semantics, but the benefits of providing our tracks with this additional information are clear and, for some applications, essential. Therefore, we use labels (that we call *specification positions*) to uniquely identify each literal in a specification which roughly corresponds to nodes in the CSP specification's abstract syntax tree. We define a function $\mathcal{P}os$ to obtain the specification position of an element of a CSP specification and it is defined over nodes of an abstract syntax tree for a CSP specification. Formally,

Definition 1. (Specification position) A *specification position* is a pair (N, w) where $N \in \mathcal{N}$ and w is a sequence of natural numbers (we use Λ to denote the empty sequence). We let $\mathcal{P}os(o)$ denote the specification position of an expression o. Each process definition $N = P$ of a CSP specification is labelled with specification positions. The specification position of its left-hand side is $\mathcal{P}os(N) = (N, 0)$. The right-hand side is labelled with the call $\texttt{AddSpPos}(P, (N, \Lambda))$; where function `AddSpPos` is defined as follows:

$\texttt{AddSpPos}(P, (N, w)) =$

$$\begin{cases} P_{(N,w)} & \text{if } P \in \mathcal{N} \\ STOP_{(N,w)} & \text{if } P = STOP \\ SKIP_{(N,w)} & \text{if } P = SKIP \\ a_{(N,w.1)} \rightarrow_{(N,w)} \texttt{AddSpPos}(Q,(N,w.2)) & \text{if } P = a \rightarrow Q \\ \texttt{AddSpPos}(Q,(N,w.1))\ op_{(N,w)}\ \texttt{AddSpPos}(R,(N,w.2)) & \text{if } P = Q\ op\ R\ \ \forall\ op \in \{\sqcap, \Box, ||, ;\} \end{cases}$$

Example 3. *Consider again the CSP specification in Example 2 where literals are labelled with its associated specification positions (they are underlined) so that labels are unique:*

$$\begin{aligned} \texttt{MAIN}_{\underline{(\texttt{MAIN},0)}} \ &= \ (\texttt{a}_{\underline{(\texttt{MAIN},1.1)}} \rightarrow_{\underline{(\texttt{MAIN},1)}} \texttt{STOP}_{\underline{(\texttt{MAIN},1.2)}}) \ \ \|_{\{\texttt{a}\}}^{\underline{(\texttt{MAIN},\Lambda)}} \\ &\quad (\texttt{P}_{\underline{(\texttt{MAIN},2.1)}} \Box_{\underline{(\texttt{MAIN},2)}} (\texttt{a}_{\underline{(\texttt{MAIN},2.2.1)}} \rightarrow_{\underline{(\texttt{MAIN},2.2)}} \texttt{STOP}_{\underline{(\texttt{MAIN},2.2.2)}})) \\ \texttt{P}_{\underline{(\texttt{P},0)}} \ &= \ \texttt{b}_{\underline{(\texttt{P},1)}} \rightarrow_{\underline{(\texttt{P},\Lambda)}} \texttt{SKIP}_{\underline{(\texttt{P},2)}} \end{aligned}$$

In the following, specification positions will be represented with greek letters $(\alpha, \beta, \ldots)$ and we will often use indistinguishably an expression and its associated specification position when it is clear from the context (e.g., in Example 3 we will refer to $(\texttt{P},\texttt{1})$ as $\texttt{b}$).

In order to introduce the formal definition of track, we need first to define the concept of *control-flow*, which refers to the order in which the individual literals of a CSP specification are executed. Intuitively, the control can pass from a specification position α to a specification position β iff an execution exists where α is executed before β. This notion of control-flow is similar of the control-flow used in the *control-flow graphs* (CFG) [17] of imperative programming. We have adapted the same idea to CSP where choices and parallel composition appears; and in a similar way to the CFG, we use this definition to draw control arcs in our tracks. Formally,

Definition 2. (Static control-flow) Given a CSP specification $\mathcal{S}$ and two specification positions α, β in $\mathcal{S}$, we say that the *control can pass from* α *to* β, denoted by $\alpha \Rightarrow \beta$, iff one of the following conditions holds:

i) $\alpha = N \ \wedge \ \beta = \mathit{first}((N,\Lambda))$ with $N = rhs(N) \in \mathcal{S}$
ii) $\alpha \in \{\sqcap, \Box, ||\} \ \wedge \ \beta \in \{\mathit{first}(\alpha.1), \mathit{first}(\alpha.2)\}$
iii) $\alpha \in \{\rightarrow, ;\} \ \wedge \ \beta = \mathit{first}(\alpha.2)$
iv) $\alpha = \beta.1 \ \wedge \ \beta = \rightarrow$
v) $\alpha \in \mathit{last}(\beta.1) \ \wedge \ \beta = \ ;$

where $\mathit{first}(\alpha)$ is the specification position of the subprocess denoted by α which must be executed first:

$$\mathit{first}(\alpha) = \begin{cases} \alpha.1 & \text{if } \alpha = \ \rightarrow \\ \mathit{first}(\alpha.1) & \text{if } \alpha = \ ; \\ \alpha & \text{otherwise} \end{cases}$$

and $\mathit{last}(\alpha)$ is the set of all possible termination points of the subprocess denoted

by α:

$$last(\alpha) = \begin{cases} \{\alpha\} & \text{if } \alpha = \texttt{SKIP} \\ \emptyset & \text{if } \alpha = \texttt{STOP} \vee \\ & (\alpha \in \{||\} \wedge (last(\alpha.1) = \emptyset \vee last(\alpha.2) = \emptyset)) \\ last(\alpha.1) \cup last(\alpha.2) & \text{if } \alpha \in \{\sqcap, \Box\} \vee \\ & (\alpha \in \{||\} \wedge last(\alpha.1) \neq \emptyset \wedge last(\alpha.2) \neq \emptyset) \\ last(\alpha.2) & \text{if } \alpha \in \{\rightarrow, ;\} \\ last((N, \Lambda)) & \text{if } \alpha = N \end{cases}$$

For instance, in Example 3, we can see how the control can pass from a specification position to another one, e.g., we have $(\texttt{MAIN}, 2) \Rightarrow (\texttt{MAIN}, 2.1)$ and $(\texttt{MAIN}, 2) \Rightarrow (\texttt{MAIN}, 2.2.1)$ due to rule ii). And $(\texttt{MAIN}, 2.2.1) \Rightarrow (\texttt{MAIN}, 2.2)$ due to rule iv); $(\texttt{MAIN}, 2.2) \Rightarrow (\texttt{MAIN}, 2.2.2)$ due to rule iii) and $(\texttt{MAIN}, 2.1) \Rightarrow (\texttt{P}, 1)$ due to rule i).

We also need to define the notions of *rewriting step* and *derivation*.

Definition 3. (Rewriting Step, Derivation) Given a CSP process P, a *rewriting step* for P, denoted by $P \overset{\Theta}{\rightsquigarrow} P'$, is the transformation of P into P' by using a rule of the CSP semantics. Therefore, $P \overset{\Theta}{\rightsquigarrow} P'$ iff a rule of the form $\dfrac{\Theta}{P \xrightarrow{e} P'}$ is applicable, where $e \in \Sigma \cup \{\tau, \checkmark\}$ and Θ is a (possibly empty) set of rewriting steps. Given a CSP process P_0, we say that the sequence $P_0 \overset{\Theta_0}{\rightsquigarrow} \ldots \overset{\Theta_n}{\rightsquigarrow} P_{n+1}$, $n \geq 0$, is a *derivation* of P_0 iff $\forall\ i, 0 \leq i \leq n, P_i \overset{\Theta_i}{\rightsquigarrow} P_{i+1}$ is a rewriting step. We say that the derivation is *complete* iff there is no possible rewriting step for P_{n+1}. We say that the derivation has *successfully finished* iff P_{n+1} is Ω.

For instance, in Fig. 5(a), one (possible) complete derivation of Example 3 is shown (for the time being, the reader can ignore the underlined part). The rules applied in each rewriting step (ignoring subderivations) are (Process Call) and (Synchronized Parallelism 3) (abbrev. (PC) and (SP3), respectively).

Function *last* of Definition 2 can be used to determine the last specification position in a derivation. However, this function computes all possible final specification positions, and a derivation only reaches (non-deterministically) a set of them. Therefore, we will use in the following a modified version of *last* called $last'$ whose behaviour is exactly the same as *last* except in the case of choices where only one of the branches is selected:

For each derivation $(P \sqcap P' \overset{\Theta}{\rightsquigarrow} P)$ or $(P \ \Box\ P' \overset{\Theta_0}{\rightsquigarrow} \ldots \overset{\Theta_n}{\rightsquigarrow} P'', n \geq 0$ such that $P \overset{\Theta'_0}{\rightsquigarrow} \ldots \overset{\Theta'_m}{\rightsquigarrow} P'', m \geq 0)$, $last'(P \sqcap P') = last'(P \ \Box\ P') = last'(P)$.

Note that, while *last* is static, $last'$ is dynamic; it is defined in the context of a particular derivation which implies one particular way of resolving any non-determinism. The same happens with the definition of control-flow. Control-flow is defined statically and says if the control can pass from α to β in some derivation. However, the track is a dynamic structure produced for a particular derivation. Therefore, we produce a dynamic version of the definition of control-flow which is defined for a particular derivation.

Definition 4. (Dynamic control-flow) Let $\mathcal{S}$ be a CSP specification and $\mathcal{D}$ a derivation in $\mathcal{S}$. Given two specification positions α, β in $\mathcal{S}$, we say that the

control can dynamically pass from α *to* β, denoted by $\alpha \Rrightarrow \beta$, iff the control can pass from α *to* β ($\alpha \Rightarrow \beta$) in derivation $\mathcal{D}$. For each $P \overset{\Theta}{\rightsquigarrow} P' \in \mathcal{D}$ and for all rewriting steps in Θ, we have that:

1. if P is a prefixing $(a \rightarrow Q)$ or a sequential composition $(Q; R)$, then $\mathcal{P}os(a) \Rrightarrow \mathcal{P}os(\rightarrow)$ or $\forall p \in last'(Q)$, $\mathcal{P}os(p) \Rrightarrow \mathcal{P}os(;)$ respectively,

2. if $P \Rightarrow first(P'')$ where $P'' \overset{\Theta'}{\rightsquigarrow} P''' \in \Theta$, then $\mathcal{P}os(P) \Rrightarrow \mathcal{P}os(first(P''))$,

3. if $P \Rightarrow first(P')$, then $\mathcal{P}os(P) \Rrightarrow \mathcal{P}os(first(P'))$.

Clauses 1, 2 and 3 define the cases in which the control passes between two specification positions in a given derivation. In clause 1, if we have a prefixing in the control then Θ is empty and the rewriting step applied is of the form $\frac{}{(a \rightarrow P) \xrightarrow{a} P}$. In this case, clause 1 guarantees that the control can dynamically pass from a to $\rightarrow$; and clause 3 guarantees that the control can dynamically pass from $\rightarrow$ to P. However, in general, Θ is not empty, and the rewriting step is of the form $\frac{P'' \longrightarrow P'''}{P \longrightarrow P'}$. Here, clause 2 ensures that the control can dynamically pass from P to P''; and clause 3 ensures that the control can dynamically pass from P to P' and from P'' to P'''. For instance, it is possible that we have a rewriting step to evaluate the process $P \,\Box\, P'$. Clearly, the control can pass from $\Box$ to both P and P' ($\Box \Rightarrow P$ and $\Box \Rightarrow P'$), but in the rewriting step the control will only pass to one of them ($\Box \Rrightarrow P$ or $\Box \Rrightarrow P'$). In this case, clauses 2 and 3 are used.

We are now in a position to formally define the concept of *track* of a derivation.

Definition 5. (Track) Given a CSP specification $\mathcal{S}$, and a derivation $\mathcal{D}$ in $\mathcal{S}$, the *track* of $\mathcal{D}$ is a graph $\mathcal{G} = (N, E_c, E_s)$ where N is a set of nodes uniquely identified with a natural number and that are labelled with specification positions ($l(n)$ refers to the *label* of node n), and edges are divided into two groups:

- *control-flow edges* (E_c) are a set of one-way edges (denoted with $\mapsto$) representing the control-flow between two nodes, and

- *synchronization edges* (E_s) are a set of two-way edges (denoted with $\leftrightarrow$) representing the synchronization of two (event) nodes;

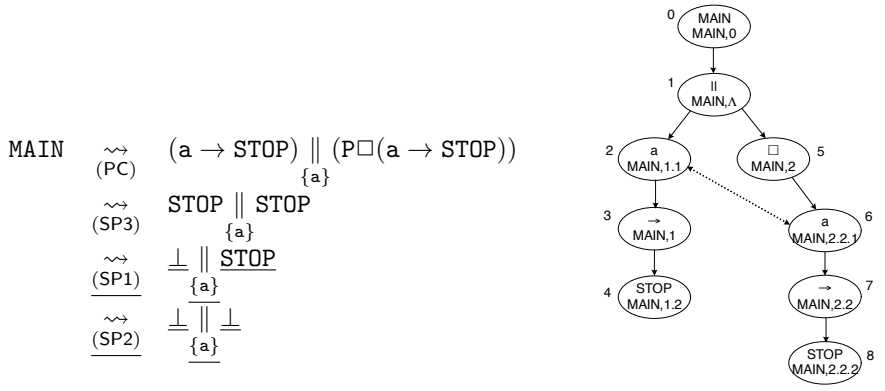
and

1. E_c contains a control-flow edge $a \mapsto a'$ iff $a \Rrightarrow a'$ with respect to $\mathcal{D}$, and

2. E_s contains a synchronization edge $a \leftrightarrow a'$ for each synchronization occurring in $\mathcal{D}$ where a and a' are the nodes of the synchronized events.

The only nodes in N are the nodes induced by E_c and E_s.

Example 4. *Consider again the specification of Example 3. We show in Fig. 5(a) one possible derivation (ignoring subderivations) of this specification (for the time being, the underlined part should be ignored). Its associated track is shown*



(a) Derivation (b) Track

Figure 5: Derivation and track associated with the specification of Example 3

in Fig. 5(b). In the example, we see that the track is a connected and directed graph. Apart from the control-flow edges, there is one synchronization edge between nodes $(\texttt{MAIN}, 1.1)$ *and* $(\texttt{MAIN}, 2.2.1)$ *representing the synchronization of event* $\mathtt{a}$*. To illustrate the inclusion of edges in Definition 5, we see that the edge between nodes* 2 *and* 3 *is introduced according to clause 1 of Definition 4; the edge between nodes* 5 *and* 6 *is introduced according to clause 2 of Definition 4 because, in the subderivations of* (SP3)*, there is a rewriting step* $\dfrac{\dfrac{}{(\mathtt{a} \to \mathit{STOP}) \xrightarrow{a} (\mathit{STOP})}\,(\text{Prefixing})}{(P\Box(\mathtt{a} \to \mathit{STOP})) \xrightarrow{a} (\mathit{STOP})}\,(\text{External Choice 4})$ *and* $\mathit{first}(\mathtt{a} \to \mathit{STOP}) = \mathtt{a}$*; the edge between nodes 7 and 8 is introduced according to clause 3 of Definition 4 because there is also a rewriting step* $\dfrac{}{(\mathtt{a} \to \mathit{STOP}) \xrightarrow{a} (\mathit{STOP})}\,(\text{Prefixing})$ *and* $\mathit{first}(\mathit{STOP}) = \mathit{STOP}$*; and the synchronization edge between nodes 2 and 6 is introduced according to clause 2 of Definition 5.*

The trace associated with the derivation in Fig. 5(a) is $\langle \mathtt{a} \rangle$*. Therefore, note that the track is much more informative: it shows the exact processes that have been evaluated with an explicit causality relation; and, in addition, it shows the specification positions that have been evaluated and in what order.*

4. Instrumenting the semantics for tracking

The generation of tracks in CSP introduces new challenges such as non-determi-nistic execution of processes, deadlocks, non-terminating processes and synchronizations. In this work, we designed a solution that overcomes these difficulties. Firstly, we generate tracks with an augmented semantics which is conservative with respect to the standard operational semantics. Therefore, the execution order is the standard order, thus non-determinism and synchronizations are solved by the own semantics. Moreover, the semantics generates the track incrementally, step by step. Therefore, infinite computations can be tracked until they are stopped. Hence, it is not needed to actually finish a computation to get the track of the subcomputations performed. In order to solve

the problem of deadlocks (that stop the computation), and have a representation for them in the tracks; when a deadlock happens, the semantics performs some additional steps to be able to generate a part of the track that represents the deadlock. These additional steps do not influence the other rules of the semantics, thus it remains conservative.

This section introduces an instrumented operational semantics of CSP which generates as a side-effect the tracks associated with the computations performed with the semantics. The tracking semantics is shown in Fig. 6, where we assume that every literal in the program has been labelled with its specification position (denoted by a subscript, e.g., P_α). In this semantics, a *state* is a tuple (P, G, m, Δ), where P is the process to be evaluated (the *control*), G is a directed graph (i.e., the track built so far), m is a numeric reference to the current node in G, and Δ is a set of references to nodes that may be synchronized. Concretely, m references the node in G where the specification position of the control P must be stored. Reference m is a fresh[2] reference generated to add new nodes to G. The basic idea of the graph construction is to record the current control with the current reference in every step by connecting it to its parent. We use the notation $G[m \underset{n}{\mapsto} \alpha]$ to introduce a node in G. For instance, if we are adding a node to G this new node has reference m, it is labelled with specification position α, and its successor is n (a fresh reference). Successor arrows are denoted by $m \underset{n}{\mapsto}$ which means that node n is the successor of node m. Every time an event in Σ happens during the computation, this event is stored in the set Δ of the current state. Therefore, when a synchronized parallelism is evaluated, all the events that must be synchronized are in Δ. We use the special symbol $\perp$ to denote any process that is deadlocked. In order to perform computations, we construct an initial state (e.g., $(\texttt{MAIN}, \emptyset, 0, \emptyset)$) and (non-deterministically) apply the rules of Fig. 6. When the execution has finished or has been interrupted, the semantics has produced the track of the computation performed so far.

An explanation for each rule of the semantics follows:

(Process Call) The called process N is unfolded, node m is added to the graph with specification position α and successor n (a fresh reference). The new process in the control is $rhs(N)$. The set Δ of events to be synchronized is put to $\emptyset$.

(Prefixing) This rule adds nodes m (the prefix) and n (the prefixing operator) to the graph. In the new state, n becomes the parent reference and the fresh reference p represents the current reference. The new control is P. The set Δ is $\{m\}$ to indicate that event a has occurred and it must be synchronized when required by (Synchronized Parallelism 3).

(SKIP and STOP) Whenever one of these rules is applied, the subcomputation finishes because Ω (for rule `SKIP`) and $\perp$ (for rule `STOP`) are put in the control, and these special symbols have no associated rule. A node with the `SKIP` (respectively `STOP`) specification position is added to the graph.

(Internal Choice 1 and 2) The choice operator is added to the graph, and the (non-deterministically) selected branch is put into the control with the

[2]We assume that fresh references are numeric and generated incrementally.

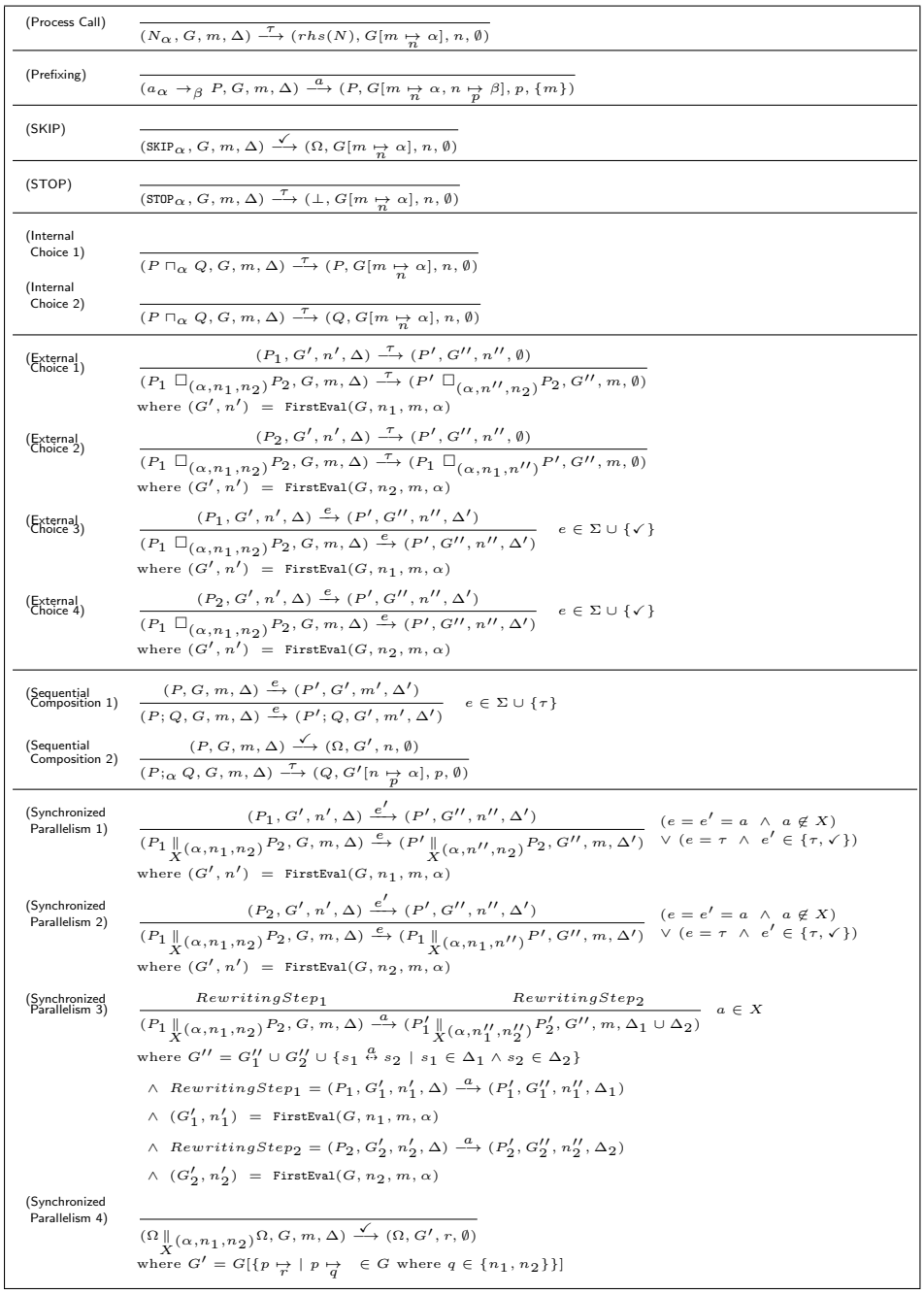


Figure 6: An instrumented operational semantics to generate CSP tracks

fresh reference n as the successor of the choice operator.

(External Choice 1, 2, 3 and 4) External choices can develop both branches while τ events happen (rules 1 and 2), until an event in $\Sigma \cup \{\checkmark\}$ occurs (rules 3 and 4). This means that the semantics can add nodes to both branches of

the track alternatively, and thus, it needs to store the next reference to use in every branch of the choice. This is done by labelling choice operators with a tuple of the form (α, n_1, n_2) where α is the specification position of the choice operator; and n_1 and n_2 are respectively the references to be used in the left and right branches of the choice, and they are initialized to $\bullet$, a symbol used to express that the branch has not been evaluated yet. Therefore, the first time a branch is evaluated, we generate a new reference for this branch. For this purpose, function `FirstEval` is used:

$$\texttt{FirstEval}(G, n, m, \alpha) \;=\; \begin{cases} (G[m \underset{p}{\mapsto} \alpha], p) & \text{if } n = \bullet \\ (G, n) & \text{otherwise} \end{cases}$$

This function checks whether this is the first time that the branch is evaluated (this only happens when the reference of this branch is empty, i.e., $n = \bullet$). In this case, the choice operator is added to G. For instance, consider the rewriting step (EC4) of Fig. 7. The choice operator in the rewriting step R is labelled with $((\texttt{MAIN}, \Lambda), \bullet, \bullet)$. Therefore, it is evaluated for the first time, and thus, in the left-hand side state of the upper rewriting step, node $5 \underset{6}{\mapsto} (\texttt{MAIN}, 2)$, which refers to the choice operator, is added to G.

(Sequential Composition 1 and 2) Sequential Composition 1 is used to evolve process P until it is finished. P is evolved to P' which is put into the control. When P successfully finishes (it becomes Ω), $\checkmark$ happens. Then, Sequential Composition 2 is used and Q is put into the control. The sequential composition operator ; is added to the graph with successor p that is the reference to be used in the first node added in the subderivation associated with Q.

(Synchronized Parallelism 1 and 2) In a synchronized parallel composition, both parallel processes can be intertwiningly executed until a synchronized event is found. Therefore, nodes from both processes can be added interwoven to the graph. Hence, each parallelism operator is labelled with a tuple of the form (α, n_1, n_2) as it happens with external choices.

These rules develop the branches of the parallelism until they are finished or until they must synchronize. In order to introduce the parallelism operator into the graph, function `FirstEval` is used, as it happens in the external choice rules. For instance, consider the rewriting step (Synchronized Parallelism 3) of Fig. 7. The parallelism operator in the rewriting step *State 1* is labelled with $((\texttt{MAIN}, \Lambda), \bullet, \bullet)$. Therefore, it is evaluated for the first time, and thus, in the left-hand side state of the rewriting step L, node $1 \underset{2}{\mapsto} (\texttt{MAIN}, \Lambda)$, which refers to the parallelism operator, is added to G.

(Synchronized Parallelism 3) This rule is used to synchronize the parallel processes. In this case, both branches must perform a rewriting step with the same visible (and synchronized) event. Each branch derivation has a non-empty set of events (Δ_1, Δ_2) to be synchronized (note that this is a set because many parallelisms could be nested). Then, all references in

the sets Δ_1 and Δ_2 are mutually linked with synchronization edges. Both sets are joined to form the new set of synchronized events.

(Synchronized Parallelism 4) It is used when none of the parallel processes can proceed because they already successfully finished. In this case, the control becomes Ω indicating the successful termination of the synchronized parallelism. In the new state, the new (fresh) reference is r. This rule also adds to the graph the arcs from all the parents of the last references of each branch (n_1 and n_2) to r. Here, we use the notation $p \underset{r}{\mapsto}$ to add an edge from p to r. Note that the fact of generating the next reference in each rule allows (Synchronized Parallelism 4) to connect the final node of both branches to the next node. This simplifies other rules such as (Sequential Composition) that already has the reference of the node ready.

We illustrate this semantics with a simple example[3].

Example 5. *Consider again the specification in Example 3. Figure 5(a) shows one possible derivation (excluding subderivations) for this example. Note that the underlined part corresponds to the additional rewriting steps performed by the tracking semantics. This derivation corresponds to the execution of the instrumented semantics with the initial state* $(\texttt{MAIN}, \emptyset, 0, \emptyset)$ *shown in Fig. 7. Here, for clarity, each computation step is labelled with the applied rule; in each state, G denotes the current graph. This computation corresponds to the execution of the right branch of the choice (i.e.,* $\texttt{a} \rightarrow \texttt{STOP}$*). The final state is* $(\bot \underset{\{\texttt{a}\}}{\parallel}_{((\texttt{MAIN},\Lambda),9,10)} \bot, G', 1, \emptyset)$*. The final track* G' *computed for this execution is depicted in Fig. 5(b) where we can see that nodes are numbered with the references generated by the instrumented semantics. Note that nodes 9 and 10 were prepared by the semantics (edges to them were produced) but never used because the subcomputations were stopped in* `STOP`*. Note also that the track contains all the parts of the specification executed by the semantics. This means that if the left branch of the choice had been developed (i.e., unfolding the call to* `P`*, thus using rule (External Choice 3)), this branch would also belong to the track.*

5. Correctness

In this section we prove the correctness of the tracking semantics (in Fig. **??**) by showing that (i) the computations performed by the tracking semantics are equivalent to the computations performed by the standard semantics; and (ii) the graph produced by the tracking semantics is the track of the derivation. We also prove that the trace of a derivation can be automatically extracted from the track of this derivation.

The first theorem shows that the computations performed with the tracking semantics are all and only the computations performed with the standard semantics. The only difference between them from an operational point of view

[3]We refer the reader to Appendix A where another example is discussed.

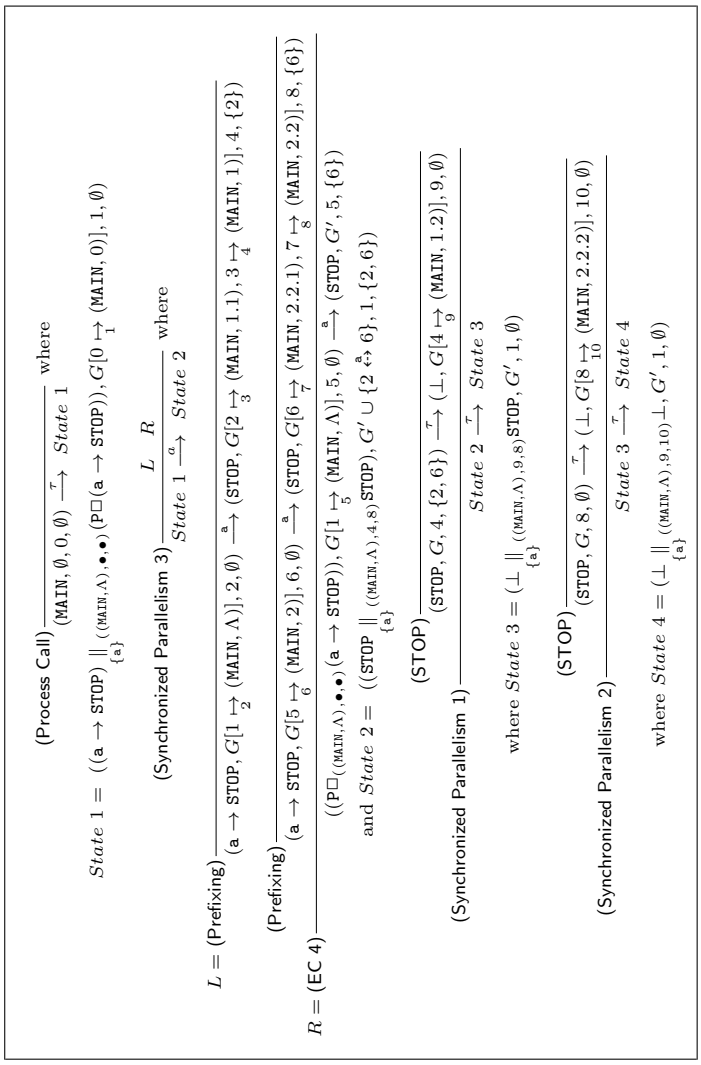


Figure 7: An example of computation with the tracking semantics in Fig. 6

is that the tracking semantics needs to perform one step when a `STOP` is evaluated (to add its specification position to the track) and then finishes, while the standard semantics finishes without performing any additional step.

Theorem 6 (Conservativeness). *Let $\mathcal{S}$ be a CSP specification, P a process in $\mathcal{S}$, and $\mathcal{D}$ and $\mathcal{D}'$ the derivations of P performed with the standard semantics of CSP and with the tracking semantics, respectively. Then, the sequence of rules applied in $\mathcal{D}$ and $\mathcal{D}'$ is exactly the same except that $\mathcal{D}'$ performs one rewriting step more than $\mathcal{D}$ for each (sub)computation that finishes with `STOP`.*

Proof. Firstly, rule (STOP) of the tracking semantics is the only rule that is not present in the standard semantics. When a (STOP) is reached in a derivation, the standard semantics stops the (sub)computation because no rule is applicable. In the tracking semantics, when a `STOP` is reached in a derivation, the only rule

applicable is (STOP) which performs τ and puts $\bot$ in the control:

$$\overline{(\texttt{STOP}_\alpha, G, m, \Delta) \xrightarrow{\tau} (\bot, G[m \underset{n}{\mapsto} \alpha], n, \emptyset)}$$

Then, the (sub)computation is stopped because no rule is applicable for $\bot$. Therefore, when the control in the derivation is STOP, the tracking semantics performs one additional rewriting step with rule (STOP).

The claim follows from the fact that both semantics have exactly the same number of rules except for rule (STOP), and these rules have the same control in all the states of the rules (thus the tracking semantics is a conservative extension of the standard semantics). Therefore, all derivations in both semantics have exactly the same number of steps and they are composed of the same sequences of rewriting steps except for (sub)derivations finishing with STOP that perform one rewriting step more (applying rule (STOP)).

The second theorem states the correctness of the tracking semantics by ensuring that the graph produced is the track of the computation. To prove this theorem, the following lemmas (proven in Appendix B) are used.

Lemma 1. *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a complete derivation of $\mathcal{S}$ performed with the tracking semantics, and $\mathcal{G}$ the graph produced by $\mathcal{D}$. Then, for each prefixing $(a \rightarrow P)$ in the control of the left state of a rewriting step in $\mathcal{D}$, we have that $\mathcal{P}os(a)$ and $\mathcal{P}os(\rightarrow)$ are nodes of $\mathcal{G}$ and $\mathcal{P}os(\rightarrow)$ is the successor of $\mathcal{P}os(a)$.*

Lemma 2. *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a complete derivation of $\mathcal{S}$ performed with the tracking semantics, and $\mathcal{G}$ the graph produced by $\mathcal{D}$. Then, for each sequential composition $(P;Q)$ in the control of the left state of a rewriting step in $\mathcal{D}$, we have that $last'(P)$ and $\mathcal{P}os(;)$ are nodes of $\mathcal{G}$ and $\mathcal{P}os(;)$ is the successor of all the elements of the set $last'(P)$ whenever P has successfully finished.*

Lemma 3. *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a complete derivation of $\mathcal{S}$ performed with the tracking semantics, and $\mathcal{G}$ the graph produced by $\mathcal{D}$. Then, for each rewriting step in $\mathcal{D}$ of the form $R_i \overset{\Theta_i}{\rightsquigarrow} R_{i+1}$ we have that:*

1. *E_c contains an edge $\mathcal{P}os(R_i) \mapsto \mathcal{P}os(first(R'))$ where $R' \overset{\Theta'}{\rightsquigarrow} R'' \in \Theta_i$ and $R_i \Rightarrow first(R')$, and*
2. *if $R_i \Rightarrow first(R_{i+1})$ then E_c contains an edge $\mathcal{P}os(R_i) \mapsto \mathcal{P}os(first(R_{i+1}))$.*

Lemma 4. *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a derivation of $\mathcal{S}$ performed with the tracking semantics, and $\mathcal{G}$ the graph produced by $\mathcal{D}$. Then, there exists a synchronization edge $(a \leftrightarrow a')$ in $\mathcal{G}$ for each synchronization in $\mathcal{D}$ where a and a' are the nodes of the synchronized events.*

Theorem 7 (Semantics correctness). *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a derivation of $\mathcal{S}$ performed with the tracking semantics, and $\mathcal{G}$ the graph produced by $\mathcal{D}$. Then, $\mathcal{G}$ is the track associated with $\mathcal{D}$.*

Proof. In order to prove that $\mathcal{G} = (N, E_c, E_s)$ is a track, we need to prove that it satisfies the properties of Definition 5. For each $R \overset{\Theta}{\rightsquigarrow} R' \in \mathcal{D}$ and for all rewriting steps in Θ we have

1. E_c contains a control-flow edge $a \mapsto a'$ iff $a \Rrightarrow a'$ with respect to $\mathcal{D}$. This is ensured by the three clauses of Definition 4:
   - by Lemma 1, if R is a prefixing $(a \to P)$, then E_c contains an edge $\mathcal{P}os(a) \mapsto \mathcal{P}os(\to)$;
   - by Lemma 2, if R is a sequential composition $(Q; P)$, then E_c contains an edge $\forall p \in \mathit{last}'(Q), \mathcal{P}os(p) \mapsto \mathcal{P}os(;)$;
   - by Lemma 3, if $R \Rightarrow \mathit{first}(R'')$ where $R'' \overset{\Theta'}{\rightsquigarrow} R''' \in \Theta$, then E_c contains an edge $\mathcal{P}os(R) \mapsto \mathcal{P}os(\mathit{first}(R''))$; and if $R \Rightarrow \mathit{first}(R')$ then E_c contains an edge $\mathcal{P}os(R) \mapsto \mathcal{P}os(\mathit{first}(R'))$; and
2. by Lemma 4, E_s contains a synchronization edge $a \leftrightarrow a'$ for each synchronization occurring in the rewriting step where a and a' are the synchronized events.

Moreover, we know that the only nodes in N are the nodes induced by E_c and E_s because all the nodes inserted in $\mathcal{G}$ are inserted by connecting the new node to the last inserted node (i.e., if the current reference is m and the new fresh reference is n, then the new node is always inserted as $G[m \underset{n}{\mapsto} \alpha]$). Hence, all nodes are related by control or synchronization edges and thus the claim holds.

Our last result states that the trace of a derivation can be extracted from its associated track. To prove it, we define first an order on the event nodes of a track that corresponds to the order in which they were generated by the tracking semantics.

Definition 6. (Event node order) Given a track $\mathcal{G} = (N, E_c, E_s)$ and nodes $m, n \in N$ such that $l(m), l(n) \in \Sigma$, m is *smaller* than n, represented by $m \ll n$ iff $m' < n'$ where $(m, m'), (n, n') \in E_c$.

Intuitively, an event node m is smaller than an event node n if and only if the successor of m has a reference smaller than the reference of the successor of n. The following lemma is also necessary to prove that the order in which events occur in a derivation is directly related with the order of Definition 6. In the following we consider an augmented version of derivation $\mathcal{D}$ which includes the event fired by the application of the rule. So, we can represent derivation $\mathcal{D}$ as $P_1 \overset{\Theta_1}{\underset{e_1}{\rightsquigarrow}} \ldots \overset{\Theta_j}{\underset{e_j}{\rightsquigarrow}} P_{j+1}$.

Lemma 5. *Given a derivation* $\mathcal{D} = P_1 \overset{\Theta_1}{\underset{e_1}{\rightsquigarrow}} \ldots \overset{\Theta_j}{\underset{e_j}{\rightsquigarrow}} P_{j+1}$ *of the tracking semantics, and the track* $\mathcal{G} = (N, E_c, E_s)$ *produced by* $\mathcal{D}$*, then* $\forall e_i \in \Sigma, 1 \leq i \leq j$*,*

- $\exists n \in N$ *such that* $l(n) = e_i$*, and*
- $\exists (n, n') \in E_c$ *such that* $n' = n + 1$*.*

Therefore, Lemma 5 ensures that the order of Definition 6 corresponds to the order in which the semantics generates the nodes, because each event is added to the graph together with a new fresh reference for the prefixing operator. Since references are generated incrementally, the occurrence of an event e will

generate a reference which is less than the reference generated with a posterior event e'. With this order, we can easily define a transformation to extract a trace from a track based on the following proposition:

Proposition 1. *Given a track $\mathcal{G} = (N, E_c, E_s)$, the trace induced by $\mathcal{G}$ is the sequence of events $T = e_1, \ldots, e_m$ that labels the associated sequence of nodes $T' = n_1, \ldots, n_m$ (i.e., $\forall e_i \in T, n_i \in T', 1 \leq i \leq m, l(n_i) = e_i$ and $e_i \in \Sigma$) where:*

1. $\forall n_i \in T',\ 0 < i < m,\ n_i \ll n_{i+1}$
2. $\forall n \in N$ *such that* $l(n) \in \Sigma$, *if* $(\nexists n' \in N \mid (n, n') \in E_s)$, *then* $n \in T'$
3. $\forall n \in N$ *such that* $l(n) \in \Sigma$, *if* $(\forall n' \in N \mid (n, n') \in E_s \wedge\ n' \ll n)$, *then* $n \in T'$

The proof of this proposition can be found in Appendix B.

Theorem 8 (Track correctness). *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a derivation of $\mathcal{S}$ produced by the sequence of events (i.e., the trace) $T = e_1, \ldots, e_m$, and $\mathcal{G}$ the track associated with $\mathcal{D}$. Then, there exists a function f that extracts the trace T from the track $\mathcal{G}$, i.e., $f(\mathcal{G}) = T$.*

Proof. Proposition 1 allows to trivially define a function f such that $f(\mathcal{G}) = T$ being $\mathcal{G}$ the track of a derivation $\mathcal{D}$, and being T the trace of the same derivation. For a track $\mathcal{G} = (N, E_c, E_s)$ we have that

$$f((n : ns), E_c, E_s) = \begin{cases} \{f((ns), E_c, E_s)\} & \text{if } (\exists n' \in N | (n, n') \in E_s \wedge n \ll n') \\ (l(n) : f((ns), E_c, E_s)) & \text{otherwise} \end{cases}$$

where list $(n : ns)$ corresponds to the set $\{n \in N \mid l(n) \in \Sigma\}$ ordered with respect to order $\ll$ of Definition 6.

6. Conclusions

This work introduces the first semantics of CSP instrumented for tracking. Therefore, it is an interesting result because it can serve as a reference mark to define and prove properties such as completeness of static analyses which are based on tracks [13, 14, 15]. The execution of the tracking semantics produces a graph as a side effect which is the track of the computation. This track is produced step by step by the semantics, and thus, it can be also used to produce a track of an infinite computation until it is stopped. The generated track can be useful not only for tracking computations but for debugging and program comprehension. This is due to the fact that our generated track also includes the specification positions associated with the expressions appearing in the track. Therefore, tracks could be used to analyse what parts of the program are executed (and in what order) in a particular computation. Also, this information allows a track viewer tool to highlight the parts of the code that are executed in each step. Notable analyses that use tracks are [3, 4, 5, 1, 13, 14, 15]. The introduction of this semantics allows us to adapt these analyses to CSP. On the practical side, we have implemented a tool called *SOC* [13] which is able to automatically generate tracks of a CSP specification. These tracks are later used for debugging. *SOC* has been integrated into the

most extended CSP animator and model-checker ProB [11, 12], that shows the maturity and usefulness of this tool and of tracks. The implementation, source code and several examples are publicly available at: `http://users.dsic.upv.es/~jsilva/soc/`

Acknowledgements

We want to thank the anonymous referees for many valuable comments and useful suggestions.

References

[1] B. Brassel, M. Hanus, F. Huch, and G. Vidal. A Semantics for Tracing Declarative Multi-paradigm Programs. In *Proc. 6th ACM SIGPLAN Int'l Conf. on Principles and Practice of Declarative Programming (PPDP'04)*, pp. 179–190, Verona, Italy, 2004.

[2] I. Brückner and H. Wehrheim. Slicing an Integrated Formal Method for Verification. In K. K. Lau, R. Banach, editors, *Int'l Conf. on Formal Engineering Methods (ICFEM'05)*, LNCS 3785, Springer-Verlag, pp. 360–374, 2005.

[3] O. Chitil. A Semantics for Tracing. In *13th Int'l Workshop on Implementation of Functional Languages (IFL'01)*, pp. 249–254, Ericsson CSL, 2001.

[4] O. Chitil, C. Runciman, and M. Wallace. Transforming Haskell for Tracing. In *14th Int'l Workshop on Implementation of Functional Languages (IFL'02)*, Revised Selected Papers, LNCS 2670, pp. 165–181, 2003.

[5] O. Chitil and Y. Lou. Structure and Properties of Traces for Functional Programs. *Electronic Notes in Theoretical Computer Science (ENTCS)*, vol. 176(1), pp. 39–63, 2007.

[6] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, vol. 9(3), pp. 319–349, 1987.

[7] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[8] K. M. Kavi, F. T. Sheldon, B. Shirazi, and A. R. Hurson. Reliability analysis of CSP specifications using Petri nets and Markov processes. In *Proc. 28th Annual Hawaii Int'l Conf. on System Sciences (HICSS'95)*, vol. 2 (Software Technology), pp. 516–524, Kihei, Maui, Hawaii, USA, 1995.

[9] J. Krinke. Context-Sensitive Slicing of Concurrent Programs. *ACM SIGSOFT Software Engineering Notes*, vol. 28(5), 2003.

[10] P. Ladkin and B. Simons. Static Deadlock Analysis for CSP-Type Communications. *Responsive Computer Systems (Chapter 5)*, Kluwer Academic Publishers, 1995.

[11] M. Leuschel and M. Butler. ProB: an automated analysis toolset for the B method. *Journal of Software Tools for Technology Transfer*, vol. 10(2), pp. 185–203, 2008.

[12] M. Leuschel and M. Fontaine. Probing the depths of CSP-M: A new FDR-compliant validation tool. In *Proc. of the 10th Int'l Conf. on Formal Engineering Methods (ICFEM'08)*, LNCS 5256, pp. 278–297, Springer-Verlag, 2008.

[13] M. Leuschel, M. Llorens, J. Oliver, J. Silva, and S. Tamarit. SOC: a Slicer for CSP Specifications. In *Proc. of the 2009 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM'09)*, pp. 165–168, Savannah, GA, USA, 2009.

[14] M. Leuschel, M. Llorens, J. Oliver, J. Silva, and S. Tamarit. The MEB and CEB Static Analysis for CSP Specifications. In *Post-proceedings of the 18th Int'l Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'08)*, Revised Selected Papers, LNCS 5438, Springer-Verlag, pp. 103–118, 2009.

[15] M. Llorens, J. Oliver, J. Silva, and S. Tamarit. An Algorithm to Generate the Context-sensitive Synchronized Control Flow Graph. To appear in *Proc. of the 25th ACM Symposium on Applied Computing (SAC 2010)*, Sierre, Switzerland, 2010.

[16] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 2005.

[17] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, vol. 3, pp. 121–189, 1995.

[18] M. D. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, vol. 10(4), pp. 352–357, 1984.

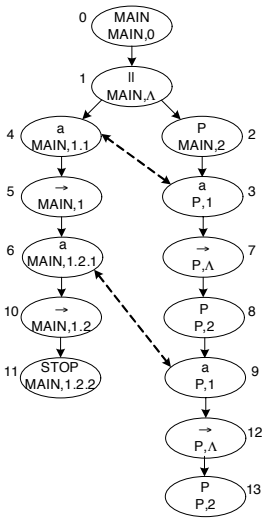


Figure 8: Track of the program in Example 9

Note for the reviewers: The following appendices have been only included to ease the reviewing process, and they will not be part of the final paper. In case of acceptance, these appendices will be published as a technical report so that the interested reader will have public access to them.

A. Tracking a specification with non-terminating processes

In this appendix we show another example whose execution finishes with a deadlock and where a non-terminating process appears.

Example 9. *Consider the following CSP specification where each (sub)process has been labelled with its associated specification position (underlining).*

$$\underline{\mathtt{MAIN}}_{\underline{(\mathtt{MAIN},0)}} = \mathtt{a}_{\underline{(\mathtt{MAIN},1.1)}} \rightarrow_{\underline{(\mathtt{MAIN},1)}} \mathtt{a}_{\underline{(\mathtt{MAIN},1.2.1)}} \rightarrow_{\underline{(\mathtt{MAIN},1.2)}} \underline{\mathtt{STOP}}_{\underline{(\mathtt{MAIN},1.2.2)}} \underset{\{\mathtt{a}\}}{\parallel}_{\underline{(\mathtt{MAIN},\Lambda)}} \underline{\mathtt{P}}_{\underline{(\mathtt{MAIN},2)}}$$

$$\underline{\mathtt{P}}_{\underline{(\mathtt{P},0)}} = \mathtt{a}_{\underline{(\mathtt{P},1)}} \rightarrow_{\underline{(\mathtt{P},\Lambda)}} \underline{\mathtt{P}}_{\underline{(\mathtt{P},2)}}$$

We use the initial state $(\mathtt{MAIN}, \emptyset, 0, \emptyset)$ *to execute the semantics and get the computation of Fig. 9. The final state is* $(\bot \underset{\{a\}}{\parallel}_{((\mathtt{MAIN},\Lambda),14,15)} (\mathtt{a} \rightarrow \mathtt{P}), G', 1, \emptyset)$. *The final track* G' *computed is the graph of Fig. 8.*

B. Proofs of technical results

In this appendix we present the proofs of the technical results of the paper.

Theorem 1 (Conservativeness). *Let* $\mathcal{S}$ *be a CSP specification,* P *a process in* $\mathcal{S}$*, and* $\mathcal{D}$ *and* $\mathcal{D}'$ *the derivations of* P *performed with the standard semantics of CSP and with the tracking semantics, respectively. Then, the sequence of rules applied in* $\mathcal{D}$ *and* $\mathcal{D}'$ *is exactly the same except that* $\mathcal{D}'$ *performs one rewriting step more than* $\mathcal{D}$ *for each (sub)computation that finishes with* `STOP`*.*

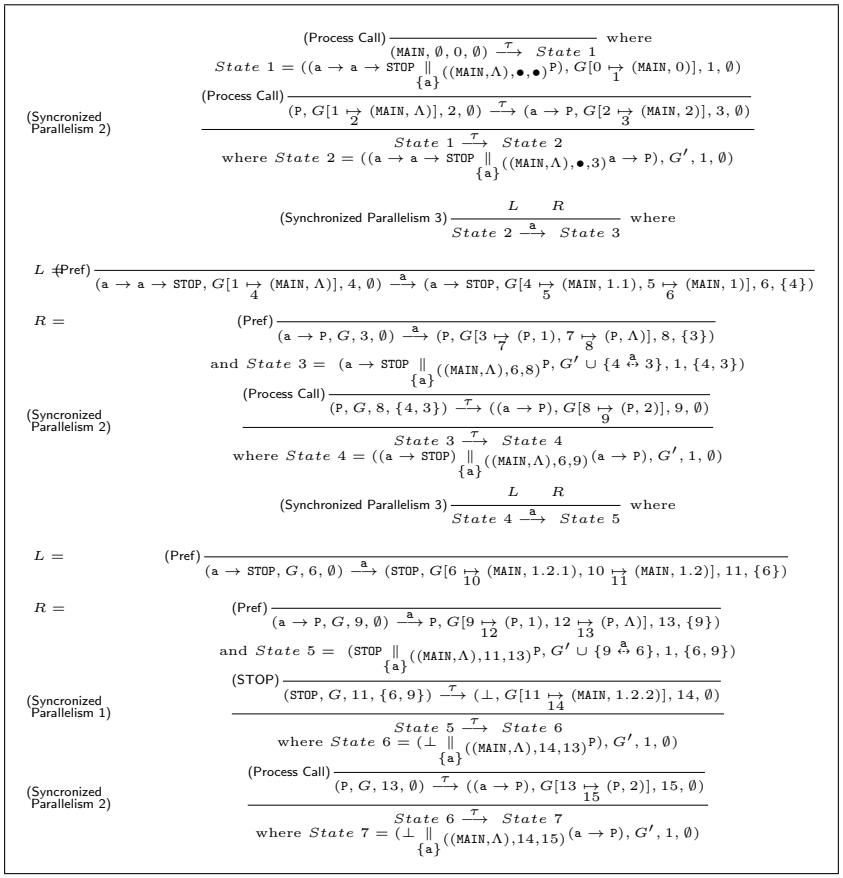


Figure 9: Computation of Example 9 with the tracking semantics in Fig. 6

PROOF. Firstly, rule (STOP) of the tracking semantics is the only rule that is not present in the standard semantics. When a (STOP) is reached in a derivation, the standard semantics stops the (sub)computation because no rule is applicable. In the tracking semantics, when a STOP is reached in a derivation, the only rule applicable is (STOP) which performs τ and puts $\bot$ in the control:

$$\frac{}{(\texttt{STOP}_\alpha, G, m, \Delta) \xrightarrow{\tau} (\bot, G[m \underset{n}{\mapsto} \alpha], n, \emptyset)}$$

Then, the (sub)computation is stopped because no rule is applicable for $\bot$. Therefore, when the control in the derivation is STOP, the tracking semantics performs one additional rewriting step with rule (STOP).

The claim follows from the fact that both semantics have exactly the same number of rules except for rule (STOP), and these rules have the same control in all the states of the rules (thus the tracking semantics is a conservative extension of the standard semantics). Therefore, all derivations in both semantics have exactly the same number of steps and they are composed of the same sequences of rewriting steps except for (sub)derivations finishing with STOP that perform one rewriting step more (applying rule (STOP)).

Now we prove Theorem 7.

Theorem 2 (Semantics correctness). *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a derivation of $\mathcal{S}$ performed with the tracking semantics, and $\mathcal{G}$ the graph produced by $\mathcal{D}$. Then, $\mathcal{G}$ is the track associated with $\mathcal{D}$.*

In order to prove the correctness of the semantics, the following lemmas are used.

Lemma 1. *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a complete derivation of $\mathcal{S}$ performed with the tracking semantics, and $\mathcal{G}$ the graph produced by $\mathcal{D}$. Then, for each prefixing $(a \rightarrow P)$ in the control of the left state of a rewriting step in $\mathcal{D}$, we have that $\mathcal{P}os(a)$ and $\mathcal{P}os(\rightarrow)$ are nodes of $\mathcal{G}$ and $\mathcal{P}os(\rightarrow)$ is the successor of $\mathcal{P}os(a)$.*

PROOF. If a prefixing $a \rightarrow P$ is in the control of the left state of a rewriting step, following the tracking semantics, only the rule (Prefixing) can be applied. By definition of this rule, m is the reference of the current node in $\mathcal{G}$. The rule (Prefixing) adds two new nodes to the graph: n and p. The node m is labelled with the specification position of event a and has successor n. The node n is labelled with the specification position of operator $\rightarrow$ and has parent m and successor p (a fresh reference). Therefore, we have that $\mathcal{P}os(a)$ and $\mathcal{P}os(\rightarrow)$ are nodes of $\mathcal{G}$ and $\mathcal{P}os(\rightarrow)$ is the successor of $\mathcal{P}os(a)$.

Lemma 6. *Given a derivation $\mathcal{D}$, for each rewriting step $R \overset{\Theta}{\rightsquigarrow} R'$ in $\mathcal{D}$ with $R' \neq \Omega$ and $R' \neq \bot$, $last'(R) = last'(R')$.*

PROOF. We prove this lemma by induction on the length of Θ. In the base case, Θ is empty, and thus only the rules (Process Call), (Prefixing), (Internal Choice 1 and 2) and (Synchronized Parallelism 4) can be applied. In all cases, the lemma holds trivially by the definition of *last'*. We assume as the induction hypothesis that the lemma holds for a non-empty Θ_i with $i > 0$ rewriting steps; and prove that the lemma also holds for a Θ_{i+1} with $i+1$ rewriting steps. We can assume that $\Theta_{i+1} = R \overset{\Theta'}{\rightsquigarrow} R'$, thus, we have to prove that the lemma holds for any possible R and R'. The possible cases are the following:

(External Choice 1 and 2) This case is trivial because the specification positions of R and R' are the same. Hence, $last'(R) = last'(R')$.

(External Choice 3 and 4) Both cases are similar. Thus, we only discuss (External Choice 3). In the case of (External Choice 3), $last'(P_1 \,\Box\, P_2) = last'(P_1)$. This rule puts P' in the control, and we know by the induction hypothesis that $last'(P_1) = last'(P')$ and, thus, the lemma holds.

(Sequential Composition 1) This case is analogous to (External Choice 1 and 2).

(Sequential Composition 2) $last'(P;Q) = last'(Q)$. Therefore the lemma holds trivially by the definition of $last'$.

(Synchronized Parallelism 1, 2 and 3) It is the same case as (External Choice 1 and 2).

Lemma 2. *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a complete derivation of $\mathcal{S}$ performed with the tracking semantics, and $\mathcal{G}$ the graph produced by $\mathcal{D}$. Then, for each sequential composition $(P;Q)$ in the control of the left state of a rewriting step in $\mathcal{D}$, we have that $last'(P)$ and $\mathcal{P}os(;)$ are nodes of $\mathcal{G}$ and $\mathcal{P}os(;)$ is the successor of all the elements of the set $last'(P)$ whenever P has successfully finished.*

Proof. If a sequential composition $(P;Q)$ is in the left state of the control of a rewriting step, following the tracking semantics, only the rules (Sequential Composition 1) and (Sequential Composition 2) can be applied. (Sequential Composition 1) is only used to evolve process P until it is finished. The application of this rule is only possible with any event except $\checkmark$, remaining the sequential composition operator in the control. (Sequential Composition 2) can only be used when $\checkmark$ happens and thus Ω is left in the control.

Therefore, when P has successfully finished evolving to Ω and with m' as the new reference, (Sequential Composition 2) is applied. This rule adds to the graph a new node n labelled with the specification position of ; that has successor p (a fresh reference). Therefore, we have that $\mathcal{P}os(;)$ is a node of $\mathcal{G}$ and $\mathcal{P}os(;)$ is the successor of the node m'. Then, we have to prove that $last'(P)$ is a set of nodes of the graph added before P successfully finished with reference m' and its successor is n.

We prove this claim by induction on the length of the derivation $P \overset{\Theta_0}{\rightsquigarrow} \ldots \overset{\Theta_n}{\rightsquigarrow} \Omega$, $n \geq 0$. The base case happens when the last rewriting step of the derivation is done leaving Ω in the control. Only these rules can be used:

(SKIP) In this case, $m \underset{n}{\mapsto} \texttt{SKIP}$ is added to $\mathcal{G}$ and n is the new reference. Because $last'(\texttt{SKIP}) = \{\texttt{SKIP}\}$, therefore, the claim follows.

(External Choice 3) Here, $last'(P_1 \,\Box\, P_2) = last'(P_1)$. This rule puts P' in the control which is Ω by the conditions of the lemma. Therefore, there must be a SKIP, which is $last'(P)$, at the top of Θ because we know that the derivation successfully finishes and thus Θ is finite.

(External Choice 4) It is analogous to the previous case, but here $last'(P_1 \,\Box\, P_2) = last'(P_2)$.

(Synchronized Parallelism 4) $last'(P_1 \mid\mid P_2) = last'(P_1) \cup last'(P_2)$. The parents of the last nodes of P_1 and P_2 are connected to the new reference r. Therefore the claim follows.

The induction hypothesis states that for all rewriting step $R \overset{\Theta}{\rightsquigarrow} R', R' \neq \Omega$ in the derivation $Q \overset{\Theta_0}{\rightsquigarrow} \dots \overset{\Theta_n}{\rightsquigarrow} \Omega, n \geq 0$ where $P \overset{\Theta'}{\rightsquigarrow} Q \in \mathcal{D}$, $last'(P)$ is put in the control of a further rewriting step of the derivation together with its reference.

Then, we prove that this also holds for the previous rewriting step $R_0 \overset{\Theta''}{\rightsquigarrow} R$. Only rules that do not perform ✓ could be applied (because ✓ puts Ω in the control of the right state and now, we are not considering the final rewriting step).

(STOP) This rule could not be applied because it puts $\perp$ in the control. There is no rule for $\perp$ thus, if applied, P could not successfully finished.

(Process Call), (Prefixing) and (Internal Choice 1 and 2) In these rules R is put in the control of the final state together with its reference. We know by Lemma 6 that $last'(R_0) = last'(R)$ thus, the claim follows by the induction hypothesis.

(External Choice 1 and 2) Both rules keep the process in the control and the same references, thus the claim follows by the induction hypothesis.

(External Choice 3 and 4) In this case, $last'(P_1 \,\Box\, P_2) = last'(P_1)$. This rule puts P' in the control, and we know by Lemma 6 that $last'(P_1) = last'(P')$, thus the lemma holds by the induction hypothesis.

(Sequential Composition 1 and 2) We know that P successfully finished, thus (Sequential Composition 1) is applied a number of times before (Sequential Composition 2), that puts Q in the control. We know that $last'(P;Q) = last'(Q)$ thus, the claim holds by the induction hypothesis.

(Synchronized Parallelism 4) $last'(P_1||P_2) = last'(P_1) \cup last'(P_2)$. The parents of the last nodes of P_1 and P_2 are connected to the new reference r. Therefore the claim follows.

In the following lemma, we need to extend the notion of rewriting step by including the graph reference to each expression. Therefore, we will use *extended* rewriting step denoted with $(R, m) \overset{\Theta}{\rightsquigarrow} (R', m')$.

Lemma 7. *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a complete derivation of $\mathcal{S}$ performed with the tracking semantics, and $\mathcal{G}$ the graph produced by $\mathcal{D}$. Then, for each extended rewriting step in $\mathcal{D}$ of the form $(R, m) \overset{\Theta}{\rightsquigarrow} (R', m')$ which is not associated with* (Synchronized Parallelism 4) *we have that a node for $first(R)$ is added to $\mathcal{G}$ with reference m.*

Proof. We prove the lemma for each rule:

(SKIP), (STOP), (Prefixing), (Process Call), and (Internal Choice 1 and 2) A node for $first(R)$ (in these rules α) is added to $\mathcal{G}$ with reference m.

(External Choice 1, 2, 3 and 4) and (Synchronized Parallelism 1, 2 and 3) In these rules, the node associated with $first(R)$ (it is α here) could be included or not, depending on whether it has been included by a previous rewriting step. If it is already included, it is due to the specification position of the previous expression in the control is the same as R, and its associated rewriting step or a previous one has added it. If other case, function `FirstEval` is called and it includes the node for R with reference m, since the corresponding n_1 and n_2 are equal to $\bullet$.

(Sequential Composition 1 and 2) In both rules, the node for $first(R)$ (in this case $first(P)$) is included by a rewriting step in Θ. All possible rewriting steps must apply one of these previous rules, and thus, the claim recursively follows.

Lemma 3. *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a complete derivation of $\mathcal{S}$ performed with the tracking semantics, and $\mathcal{G}$ the graph produced by $\mathcal{D}$. Then, for each rewriting step in $\mathcal{D}$ of the form $R_i \overset{\Theta_i}{\rightsquigarrow} R_{i+1}$ we have that:*

1. *E_c contains an edge $\mathcal{P}os(R_i) \mapsto \mathcal{P}os(first(R'))$ where $R' \overset{\Theta'}{\rightsquigarrow} R'' \in \Theta_i$ and $R_i \Rightarrow first(R')$, and*
2. *if $R_i \Rightarrow first(R_{i+1})$ then E_c contains an edge $\mathcal{P}os(R_i) \mapsto \mathcal{P}os(first(R_{i+1}))$.*

Proof. We prove each claim separately:

1. Firstly, we know that Θ cannot be empty. Therefore, rules (SKIP), (STOP), (Process Call), (Prefixing), (Internal Choice 1 and 2) and (Synchronized Parallelism 4) could not be applied. Moreover, (Sequential Composition) could never be applied because if R_i is of the form $P;Q$, then the unique possible case is that $\mathcal{P}os(P;Q) \Rightarrow \mathcal{P}os(Q)$ (by Definition 2). And Q can only be in the control of the right state; hence, Q cannot appear in Θ. Then the only applicable rules are (External Choice) or (Synchronized Parallelism).

   Let us consider extended rewriting steps $(R', m') \overset{\Theta'}{\rightsquigarrow} (R'', m'') \in \Theta_i$. First, we have to prove that a node with the specification position of R_i is included in the graph and the reference of its successor node is put in each m' of Θ_i. In rules (External Choice) and (Synchronized Parallelism) it is done using function `FirstEval`. The references associated with the selected branches of the operator must be $\bullet$, i.e., the branches have not been developed until now in the derivation. Otherwise, by Definition 2, there is no possible control flow between R_i and R'. In this case, if the corresponding reference is $\bullet$, then `FirstEval` adds to $\mathcal{G}$ the specification position of R_i and the reference of the successor node is put in all possible m'.

2. In this case, R_i cannot be neither a `SKIP` nor a `STOP`, because $\mathcal{P}os(R_i) \not\Rightarrow \mathcal{P}os(R_{i+1})$ (Ω or $\bot$, respectively) by Definition 2. Process R_i cannot be a parallelism because $\mathcal{P}os(R_i) \not\Rightarrow \mathcal{P}os(R_{i+1})$ (itself or Ω).

   If R_i is an external choice we have two possibilities. If we apply (External Choice 1 or 2) then R_i and R_{i+1} have the same specification position and thus, by Definition 2, no control flow is possible. If we apply (External

Choice 3 or 4) the control cannot pass from R_i to R_{i+1}, because R_{i+1} is different to $first(R_i.1)$ or $first(R_i.2)$. This is due to the fact that the nodes associated with these positions have necessarily been added to $\mathcal{G}$ by the rewriting step Θ_i or by a previous rewriting step on derivation $\mathcal{D}$. Therefore, process R_i must be a process call, a prefixing, an internal choice, or a sequential composition. If it is a sequential composition, rule (Sequential Composition 1) cannot be applied because in this case R_i and R_{i+1} have the same specification position. Therefore, only (Sequential Composition 2) can be applied.

We now prove that the application of any of remaining rules (Process Call), (Prefixing), (Internal Choice 1 and 2), and (Sequential Composition 2) satisfies the property.

Let $(R_i, n_i) \overset{\Theta_i}{\rightsquigarrow} (R_{i+1}, n_{i+1})$ be an extended rewriting step. In all the rules, a node labelled α is added to $\mathcal{G}$ (except in (Prefixing) where is β) and the position of its successor is placed as n_{i+1}. Furthermore, we know by Lemma 7 that a node for $\mathcal{P}os(first(R_{i+1}))$ is included in the next rewriting step in the derivation $(R_{i+1}, n_{i+1}) \overset{\Theta_{i+1}}{\rightsquigarrow} (R_{i+2}, n_{i+2})$ having associated position n_{i+1}.

Note here again that Lemma 7 excludes rule (Synchronized Parallelism 4) but in this case both branches must be already in $\mathcal{G}$ by a previous application of (Synchronized Parallelism 1, 2 or 3).

Lemma 4. *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a derivation of $\mathcal{S}$ performed with the tracking semantics, and $\mathcal{G}$ the graph produced by $\mathcal{D}$. Then, there exists a synchronization edge $(a \leftrightarrow a')$ in $\mathcal{G}$ for each synchronization in $\mathcal{D}$ where a and a' are the nodes of the synchronized events.*

Proof. We prove this lemma by induction on the length of the derivation $\mathcal{D} = R_0 \overset{\Theta_0}{\rightsquigarrow} R_1 \overset{\Theta_1}{\rightsquigarrow} \dots \overset{\Theta_n}{\rightsquigarrow} R_{n+1}$. We can assume that the derivation starts with the initial configuration $(\texttt{MAIN}, \emptyset, 0, \emptyset)$, thus in the base case, the only rule applicable is (Process Call) and hence no synchronization is possible. We assume as the induction hypothesis that there exists a synchronization edge $(a \leftrightarrow a') \in \mathcal{G}$ for each synchronization in $R_0 \overset{\Theta_0}{\rightsquigarrow} \dots \overset{\Theta_{i-1}}{\rightsquigarrow} R_i$ with $0 < i \leq n$ and prove that the lemma also holds for the next rewriting step $R_i \overset{\Theta_i}{\rightsquigarrow} R_{i+1}$.

Firstly, only (Synchronized Parallelism 3) allows the synchronization of events. Therefore, only if R_i is a synchronizing parallelism, or if a (Synchronized Parallelism 3) is applied in Θ_i, $(a \leftrightarrow a') \in \mathcal{G}$. Then, let us consider the case where $\overset{\Theta_i}{\rightsquigarrow}$ is the application of rule (Synchronized Parallelism 3). This proof is also valid in the case where (Synchronized Parallelism 3) is applied in Θ_i. We have the following rewriting step:

$$\frac{RewritingStep_1 \qquad\qquad RewritingStep_2}{(P_1 \underset{X}{\|}_{(\alpha,n_1,n_2)} P_2, G, m, \Delta) \xrightarrow{a} (P_1' \underset{X}{\|}_{(\alpha,n_1'',n_2'')} P_2', G'', m, \Delta_1 \cup \Delta_2)} \; a \in X$$

where $G'' = G_1'' \cup G_2'' \cup \{s_1 \overset{a}{\leftrightarrow} s_2 \mid s_1 \in \Delta_1 \wedge s_2 \in \Delta_2\}$

$\wedge \; RewritingStep_1 = (P_1, G_1', n_1', \Delta) \xrightarrow{a} (P_1', G_1'', n_1'', \Delta_1)$

$\wedge \; (G_1', n_1') \; = \; \texttt{FirstEval}(G, n_1, m, \alpha)$

$\wedge \; RewritingStep_2 = (P_2, G_2', n_2', \Delta) \xrightarrow{a} (P_2', G_2'', n_2'', \Delta_2)$

$\wedge \; (G_2', n_2') \; = \; \texttt{FirstEval}(G, n_2, m, \alpha)$

Because (Prefixing) is the only rule that performs an event a without further conditions, we know that P_1 must be a prefixing operator or a process containing a prefixing operator whose prefix is a, i.e., we know that the rule applied in $RewritingStep_1$ is fired with an event a; and we know that all the rules of the semantics except (Prefixing) need to fire another rule with an event a as a condition. Therefore, at the top of the condition rules, there must be a (Prefixing). The same happens with P_2. Hence, two prefixing rules (one for P_1 and one for P_2) have been fired as a condition of this rule.

In addition, the new graph G'' contains the synchronization set $\{s_1 \overset{a}{\leftrightarrow} s_2 \mid s_1 \in \Delta_1 \wedge s_2 \in \Delta_2\}$ where Δ_1 and Δ_2 are the sets of references to the events that must synchronize in $RewritingStep_1$ and $RewritingStep_2$, respectively.

Hence, we have to prove that all and only the events (a) that must synchronize in $RewritingStep_1$ are in Δ_1. We prove this by showing that all references to the synchronized events are propagated down by all rules from the (Prefixing) in the top to the (Synchronized Parallelism 3). And the proof is analogous for $RewritingStep_2$.

The possible rules applied in $(P_1, G', n', \Delta) \xrightarrow{a} (P_1', G_1'', n_1'', \Delta_1)$ are: (Prefixing) In this case, the prefix a is added to Δ_1. (External Choice 3), (External Choice 4), (Sequential Composition 1), (Synchronized Parallelism 1), (Synchronized Parallelism 2) In these cases, the set Δ is propagated down. (Synchronized Parallelism 3) In this case, the sets Δ_1 and Δ_2 are joined and propagated down.

Therefore, all the synchronized events are in the set Δ_1 and the claim follows.

Theorem 2 (Semantics correctness). *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a derivation of $\mathcal{S}$ performed with the tracking semantics, and $\mathcal{G}$ the graph produced by $\mathcal{D}$. Then, $\mathcal{G}$ is the track associated with $\mathcal{D}$.*

Proof. In order to prove that $\mathcal{G} = (N, E_c, E_s)$ is a track, we need to prove that it satisfies the properties of Definition 5. For each $R \overset{\Theta}{\rightsquigarrow} R' \in \mathcal{D}$ and for all rewriting steps in Θ we have

1. E_c contains a control-flow edge $a \mapsto a'$ iff $a \Rrightarrow a'$ with respect to $\mathcal{D}$. This is ensured by the three clauses of Definition 4:
   - by Lemma 1, if R is a prefixing $(a \to P)$, then E_c contains an edge $\mathcal{P}os(a) \mapsto \mathcal{P}os(\to)$;
   - by Lemma 2, if R is a sequential composition $(Q; P)$, then E_c contains an edge $\forall p \in last'(Q), \mathcal{P}os(p) \mapsto \mathcal{P}os(;)$;

- by Lemma 3, if $R \Rightarrow \mathit{first}(R'')$ where $R'' \overset{\Theta'}{\rightsquigarrow} R''' \in \Theta$, then E_c contains an edge $\mathcal{P}os(R) \mapsto \mathcal{P}os(\mathit{first}(R''))$; and if $R \Rightarrow \mathit{first}(R')$ then E_c contains an edge $\mathcal{P}os(R) \mapsto \mathcal{P}os(\mathit{first}(R'))$; and

2. by Lemma 4, E_s contains a synchronization edge $a \leftrightarrow a'$ for each synchronization occurring in the rewriting step where a and a' are the synchronized events.

Moreover, we know that the only nodes in N are the nodes induced by E_c and E_s because all the nodes inserted in $\mathcal{G}$ are inserted by connecting the new node to the last inserted node (i.e., if the current reference is m and the new fresh reference is n, then the new node is always inserted as $G[m \underset{n}{\mapsto} \alpha]$). Hence, all nodes are related by control or synchronization edges and thus the claim holds.

Finally, we proof Theorem 8.

Theorem 3 (Track correctness). *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a derivation of $\mathcal{S}$ produced by the sequence of events (i.e., the trace) $T = e_1, \ldots, e_m$, and $\mathcal{G}$ the track associated with $\mathcal{D}$. Then, there exists a function f that extracts the trace T from the track $\mathcal{G}$, i.e., $f(\mathcal{G}) = T$.*

To prove this theorem, we define first an order on the event nodes of a track that corresponds to the order in which they were generated by the tracking semantics.

Definition 6. *(Event node order) Given a track $\mathcal{G} = (N, E_c, E_s)$ and nodes $m, n \in N$ such that $l(m), l(n) \in \Sigma$, m is* smaller *than n, represented by $m \ll n$ iff $m' < n'$ where $(m, m'), (n, n') \in E_c$.*

Intuitively, an event node m is smaller than an event-node n if and only if the successor of m has a reference smaller than the reference of the successor of n. The following lemma is also necessary to prove that the order in which events occur in a derivation is directly related with the order of Definition 6. In the following we consider an augmented version of derivation $\mathcal{D}$ which includes the event fired by the application of the rule. So, we can represent derivation $\mathcal{D}$ as $P_1 \overset{\Theta_1}{\underset{e_1}{\rightsquigarrow}} \ldots \overset{\Theta_j}{\underset{e_j}{\rightsquigarrow}} P_{j+1}$.

Lemma 5. *Given a derivation $\mathcal{D} = P_1 \overset{\Theta_1}{\underset{e_1}{\rightsquigarrow}} \ldots \overset{\Theta_j}{\underset{e_j}{\rightsquigarrow}} P_{j+1}$ of the tracking semantics, and the track $\mathcal{G} = (N, E_c, E_s)$ produced by $\mathcal{D}$, then $\forall e_i \in \Sigma, 1 \leq i \leq j$,*

- *$\exists n \in N$ such that $l(n) = e_i$, and*
- *$\exists (n, n') \in E_c$ such that $n' = n + 1$.*

Proof. In order to prove this lemma, we prove first that any rewriting step $P_i \overset{\Theta_i}{\underset{e_i}{\rightsquigarrow}} P_{i+1}$ in $\mathcal{D}$, $1 \leq i \leq j$, with $e_i \in \Sigma$ is a prefixing or it performs a prefixing in Θ_i. This can be easily proved by showing that the rewriting step is either a prefixing (thus $\Theta_i = \emptyset$), or Θ_i has a prefixing as the top rewriting step. We prove this by case analysis. The only possible rules applied are:

(Prefixing) If this rule is applied, $\Theta_i = \emptyset$ and the claim follows trivially.

(External Choice 3 and 4), (Synchronized Parallelism 1 and 2) In these rules Θ_i contains a single rewriting step whose event is also e_i; hence, the claim follows by the induction hypothesis.

(Sequential Composition 1) This case is completely analogous to the previous one.

(Synchronized Parallelism 3) In this case, Θ_i is formed by two different rewriting steps, and both of them are similar to the previous case.

Now, both conditions hold trivially from the fact that the prefixing rule adds n to N with label $l(n) = e_i = \alpha$, and it also adds the prefixing operator $(\rightarrow)$ to N as the successor of n.

Therefore, Lemma 5 ensures that the order of Definition 6 corresponds to the order in which the semantics generates the nodes, because each event is added to the graph together with a new fresh reference for the prefixing operator. Since references are generated incrementally, the occurrence of an event e will generate a reference which is less than the reference generated with a posterior event e'. With this order, we can easily define a transformation to extract a trace from a track based on the following proposition:

Proposition 1. *Given a track $\mathcal{G} = (N, E_c, E_s)$, the trace induced by $\mathcal{G}$ is the sequence of events $T = e_1, \ldots, e_m$ that labels the associated sequence of nodes $T' = n_1, \ldots, n_m$ (i.e., $\forall e_i \in T, n_i \in T', 1 \leq i \leq m, l(n_i) = e_i$ and $e_i \in \Sigma$) where:*

1. $\forall n_i \in T',\ 0 < i < m,\ n_i \ll n_{i+1}$
2. $\forall n \in N$ *such that* $l(n) \in \Sigma$, *if* $(\nexists n' \in N \mid (n, n') \in E_s)$, *then* $n \in T'$
3. $\forall n \in N$ *such that* $l(n) \in \Sigma$, *if* $(\forall n' \in N \mid (n, n') \in E_s \wedge\ n' \ll n)$, *then* $n \in T'$

Proof. We consider a derivation $\mathcal{D} = P_1 \overset{\Theta_1}{\underset{e'_1}{\rightsquigarrow}} \ldots \overset{\Theta_j}{\underset{e'_k}{\rightsquigarrow}} P_{j+1}$. Note that $e'_1, \ldots, e'_k \neq e_1, \ldots, e_m$ because the former contains events in $\{\tau, \checkmark\}$. Then, we have that the trace is the subsequence of $e'_1, \ldots, e'_k$ that only includes events of Σ. We will represent this subsequence with $\mathcal{E} = e'_j, \ldots, e'_{j'}$ with $0 \leq j \leq j' \leq k$. Then, we have to show that $T = \mathcal{E}$. The proposition follows trivially from the fact that the sequence T follows the order of nodes imposed by Definition 6, and this order is the same order of the events that form the sequence $\mathcal{E}$ as stated by Lemma 5.

Theorem 3 (Track correctness). *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a derivation of $\mathcal{S}$ produced by the sequence of events (i.e., the trace) $T = e_1, \ldots, e_m$, and $\mathcal{G}$ the track associated with $\mathcal{D}$. Then, there exists a function f that extracts the trace T from the track $\mathcal{G}$, i.e., $f(\mathcal{G}) = T$.*

Proof. Proposition 1 allows to trivially define a function f such that $f(\mathcal{G}) = T$ being $\mathcal{G}$ the track of a derivation $\mathcal{D}$, and being T the trace of the same derivation. For a track $\mathcal{G} = (N, E_c, E_s)$ we have that

$$f((n:ns),E_c,E_s) = \begin{cases} \{f((ns),E_c,E_s)\} & \text{if } (\exists n' \in N | (n,n') \in E_s \wedge n \ll n') \\ (l(n) : f((ns),E_c,E_s)) & \text{otherwise} \end{cases}$$

where list $(n : ns)$ corresponds to the set $\{n \in N \mid l(n) \in \Sigma\}$ ordered with respect to order $\ll$ of Definition 6.

Generating a Petri net from a CSP specification: a semantics-based method[☆]

M. Llorens, J. Oliver*, J. Silva, S. Tamarit

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Valencia, Spain

Abstract

The specification and simulation of complex concurrent systems is a difficult task due to the intricate combinations of message passing and synchronizations that can occur between the components of the system. Two of the most extended formalisms used to specify, verify and simulate such kind of systems are CSP and the Petri nets. This work introduces a new technique that allows us to automatically transform a CSP specification into an equivalent Petri net. The transformation is formally defined by instrumenting the operational semantics of CSP. Because the technique uses a semantics-directed transformation, it produces Petri nets that are closer to the CSP specification and thus easier to understand. This result is interesting because it allows CSP developers not only to graphically animate their specifications through the use of the equivalent Petri net, but it also allows them to use all the tools and analysis techniques developed for Petri nets.

Key words: Concurrent programming, CSP, Petri nets, semantics, traces.

1. Introduction

Nowadays, few computers are based on a single processor architecture. Contrarily, modern architectures are based on multiprocessor systems such as the dual-core or the quad-core; and a challenge of manufacturer companies is to increase the number of processors integrated in the same motherboard. In order to take advantage of these new hardware systems, software must be prepared to work with parallel and heterogeneous components that work concurrently. This is also a necessity of the widely generalized distributed systems, and it is the

[☆]This work has been partially supported by the Spanish *Ministerio de Economía y Competitividad (Secretaría de Estado de Investigación, Desarrollo e Innovación)* under grant TIN2008-06622-C03-02 and by the *Generalitat Valenciana* under grant PROMETEO/2011/052. S. Tamarit was partially supported by the Spanish MICINN under FPI grant BES-2009-015019.

*Corresponding author

Email addresses: mllorens@dsic.upv.es (M. Llorens), fjoliver@dsic.upv.es (J. Oliver), jsilva@dsic.upv.es (J. Silva), stamarit@dsic.upv.es (S. Tamarit)

reason why the industry invests millions of dollars in the research and development of concurrent languages that can produce efficient programs for these systems, and that can be automatically verified thanks to the development of modern techniques for the analysis and verification of such languages.

In this work we focus on two of the most important concurrent formalisms: the Communicating Sequential Processes (CSP) [10, 24] and the Petri nets [18, 20]. CSP is an expressive process algebra with a big collection of software tools for the specification and verification of complex systems. In fact, CSP is currently one of the most extended concurrent specification languages and it is being successfully used in several industrial projects [3, 8]. Complementarily, Petri nets are particularly useful for the simulation and animation of concurrent specifications. They can be used to graphically animate a specification and observe the synchronization of components step by step. For these reasons, attempts to combine both models exist (see, e.g., [1]). In this work we define a fully automatic transformation that allows us to transform a CSP specification into an equivalent Petri net (i.e., the sequences of observable events produced are exactly the same). This result is very interesting because it allows CSP developers not only to graphically animate their specifications through the use of the equivalent Petri nets, but it also allows them to use all the tools and analysis techniques developed for Petri nets. Our transformation is based on an instrumentation of the CSP's operational semantics. Roughly speaking, we define an algorithm that explores all computations of a CSP specification by using the instrumented semantics. The execution of the semantics produces as a side-effect the Petri net associated with each computation, and thus the final Petri net is produced incrementally.

In summary, the steps performed by the transformation are the following: firstly, the algorithm takes a CSP specification and executes the extended semantics with an empty store. The execution of the semantics produces a Petri net that represents the performed computation. When the computation is finished, the extended semantics returns to the algorithm a new store with the information about the choices that have been executed. Then, the algorithm determines with this information whether new computations not explored yet exist. If this is the case, the semantics is executed again with an updated store. This is repeated until all possible computations have been explored. This sequence of steps gradually augments the Petri net produced. When the algorithm detects that no more computations are possible (i.e., the store is empty), it outputs the current Petri net as the final result.

This work extends a previous work by the same authors presented at the *7th International Conference on Engineering Computational Technology* [14]. In this new version we provide additional explanations and examples, and new important original material. The new material includes:

1. An instrumentation of the standard CSP operational semantics that produces as a side-effect a Petri net associated to the computations performed with the semantics.
2. New simplification algorithms that significantly reduce the size of the Petri nets generated while keeping the equivalence properties.
3. An improved implementation that has been made public (both the source code and an online version).
4. The correctness results. They prove the termination of the transformation

algorithm; and the equivalence between the produced Petri net and the original CSP specification.

The rest of the paper has been organized as follows. Section 2 overviews related work and previous approaches to the transformation of CSP into Petri nets. In Section 3 we briefly recall the syntax and semantics of CSP and Petri nets. Section 4 presents an algorithm able to generate a Petri net equivalent to a given CSP specification. To obtain the Petri net, the algorithm uses an instrumentation of the standard operational semantics of CSP which is also introduced in this section. Then, in Section 5 we introduce some algorithms to further transform the generated Petri nets. The transformation simplifies the final Petri net producing a reduced version that is still equivalent to the original CSP specification. The correctness of the technique presented is proved in Section 6. In Section 7, we describe the *CSP2PN* tool, our implementation of the proposed technique. Finally, Section 8 concludes.

2. Related work

Transforming CSP to Petri nets is known to be useful since almost their origins, because it not only has a clear practical utility, but it also has a wide theoretical interest because both concurrent models are very different, and establishing relations between them allows us to extend results from one model to the other. In fact, the problem of transforming a CSP specification into an equivalent Petri net is complex due to the big differences that exist between both formalisms. For this reason, some previous approaches aiming to transform CSP to Petri nets have been criticized because, even though they are proved equivalent, it is hardly possible to see a relation between the generated Petri net and the initial CSP specification (i.e., when a transition of the Petri net is fired, it is not even clear to what CSP process corresponds this transition). In this respect, the transformation presented here is particularly interesting because the Petri net is generated directly from the operational semantics in such a way that each syntactic element of the CSP specification has a representation in the Petri net. And, moreover, the sequences of steps performed by the CSP semantics are directly represented in the Petri net. Hence, it is not difficult to map the animation of the Petri net to the CSP specification.

We can group all previous approaches aimed at transforming CSP to Petri nets into two major research lines. The first line is based on traces describing the behavior of the system. In [16], starting from a trace-based representation of the behavior of the system, according to a subset of the Hoare's theory where no sequential composition with recursion is allowed, a stochastic Petri net model is built in a modular and systematic way. The overall model is built by modeling the system's components individually, and then putting them together by means of superposition. The second line of research includes all methodologies that translate CSP specifications into Petri nets directly from the CSP syntax. One of the first works translating CSP to Petri nets was [4], where distributed termination is assumed but nesting of parallel commands is not allowed. In [6], a CSP-like language is considered and translated into a subclass of Pr/T nets with individual tokens, where neither nesting of parallel commands is allowed nor distributed termination is taken into account. Other papers in this area are [19] that considers a subset of CCSP (the union of Milner's CCS[17] and

Hoare's CSP[10]), and [5] which provides full CSP with a truly concurrent and distributed operational semantics based on Condition/Event Systems. There are also some works that translate process algebras into stochastic or timed Petri nets in order to perform real-time analyses and performance evaluation. Notable examples are [25, 15] that translate CSP specifications and [23] that define a compositional stochastic Petri net semantics for the stochastic process algebra PEPA [9]. Even though this work is essentially different from ours because it is based on different formalisms, its implementation [2] is somehow similar to ours because the translation from PEPA to stochastic Petri nets is completely automatic. As in our work, all these papers do not allow recursion of nested parallel processes because the set of places of the generated Petri net would be infinite. In some way, our new semantics-based approach opens a third line of research where the transformation is directed by the semantics.

3. CSP and Petri nets

3.1. The syntax and semantics of CSP

This section recalls CSP's syntax and operational semantics. For concretion, and to facilitate the understanding of the following definitions and algorithms, we have selected a subset of CSP that is sufficiently expressive to illustrate the method, and it contains the most important operators that produce the challenging problems such as deadlocks, non-determinism and parallel execution.

		Domains		
		$M, N \ldots \in Names$	(Process names)	
		$P, Q \ldots \in Procs$	(Processes)	
		$a, b \ldots \in \Sigma$	(Events)	
S	::=	$D_1 \ldots D_m$	(Entire specification)	
D	::=	$N = P$	(Process definition)	
P	::=	M	(Process call)	
	|	$a \rightarrow P$	(Prefixing)	
	|	$P \sqcap Q$	(Internal choice)	
	|	$P \,\Box\, Q$	(External choice)	
	|	$P \underset{X}{\parallel} Q$	(Synchronized parallelism)	$X \subseteq \Sigma$
	|	$STOP$	(Stop)	

Figure 1: Syntax of CSP specifications

Figure 1 summarizes the syntax constructions used in CSP [10] specifications. A *specification* is a finite collection of process definitions. The left-hand side of each definition is the name of a process, which is defined in the right-hand side (abbrev. *rhs*) by means of an expression that can be a call to another process or a combination of the following operators:

Prefixing ($a \rightarrow P$) Event a must happen before process P.

Internal choice ($P \sqcap Q$) The system non-deterministically chooses to execute one of the two processes P or Q.

External choice ($P \,\Box\, Q$) It is identical to internal choice but the choice comes from outside the system (e.g., the user).

Synchronized parallelism ($P \underset{X \subseteq \Sigma}{||} Q$) Both processes are executed in parallel with a set X of synchronized events. A particular case of parallel execution is *interleaving* (represented by $|||$) where no synchronizations exist (i.e., $X = \emptyset$) and thus both processes can execute in any order. Whenever a synchronized event $a \in X$ happens in one of the processes, it must also happen in the other at the same time. Whenever the set of synchronized events is not specified, it is assumed that processes are synchronized in all common events.
Stop ($STOP$) Synonym of deadlock, i.e., it finishes the current process.

Example 1. *Consider the Moore machine [11] in Figure 2 to compute the remainder of a binary number divided by three. The different values for the possible remainders are 0, 1 and 2. Note that if a decimal value n written in binary is followed by a 0 then its decimal value becomes $2n$ and if n is followed by a 1 then its value becomes $2n + 1$. If the remainder of $n/3$ is r, then the remainder of $2n/3$ is $2r$ mod 3. If $r = 0$, 1, or 2, then $2r$ mod 3 is 0, 2, or 1, respectively. Similarly, the remainder of $(2n + 1)/3$ is 1, 0, or 2, respectively. So, this machine has 3 states: q_0 is the start state and represents a remainder 0, state q_1 represents a remainder 1 and state q_2 represents a remainder 2.*

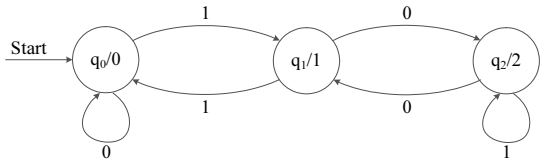


Figure 2: Moore machine to determine the remainder of a binary number divided by three

The following CSP specification corresponds to the previous Moore machine with a slight modification: the fact that a binary number is divisible by 3 is explicitly represented. Processes `REM0`, `REM1` *and* `REM2` *are considered as remainder 0, 1 and 2 states, respectively. We know that a number n is divisible by 3 if the remainder of $n/3$ is 0. So, process* `REM0` *also represents that the number is divisible by 3 (represented with the event* `divisible3`*).*

$$\texttt{MAIN} = \texttt{REM0}$$
$$\texttt{REM0} = (\texttt{0} \rightarrow \texttt{REM0}) \ \Box \ (\texttt{1} \rightarrow \texttt{REM1}) \ \Box \ (\texttt{divisible3} \rightarrow \texttt{STOP})$$
$$\texttt{REM1} = (\texttt{0} \rightarrow \texttt{REM2}) \ \Box \ (\texttt{1} \rightarrow \texttt{REM0})$$
$$\texttt{REM2} = (\texttt{0} \rightarrow \texttt{REM1}) \ \Box \ (\texttt{1} \rightarrow \texttt{REM2})$$

Let us consider now another example that contains two processes running in parallel and illustrates the use of synchronizations.

Example 2. *The following CSP specification is an extension of the previous CSP specification to check whether a given binary number is divisible by 3. Process* `BINARY` *represents a binary number; in this case, the binary number* 110 *(which corresponds to the decimal value* 6*). Processes* `REM0` *and* `BINARY` *are executed in parallel with* $\{\texttt{0}, \texttt{1}, \texttt{divisible3}\}$ *as the set of synchronized events, i.e., whenever one of these synchronized events happens in process* `REM0`*, it must also happen in process* `BINARY` *at the same time, and vice versa. So, if event* `divisible3` *occurs, it means that the binary number is divisible by 3. When the binary number is not divisible by 3, the remainder of its division between 3*

will be 1 or 2 (processes `REM1` *or* `REM2`*), and then event* `divisible3` *will never happen.*

$$\texttt{MAIN} = \texttt{REM0} \underset{\{0,1,\texttt{divisible3}\}}{\parallel} \texttt{BINARY}$$
$$\texttt{REM0} = (0 \rightarrow \texttt{REM0}) \,\Box\, (1 \rightarrow \texttt{REM1}) \,\Box\, (\texttt{divisible3} \rightarrow \texttt{STOP})$$
$$\texttt{REM1} = (0 \rightarrow \texttt{REM2}) \,\Box\, (1 \rightarrow \texttt{REM0})$$
$$\texttt{REM2} = (0 \rightarrow \texttt{REM1}) \,\Box\, (1 \rightarrow \texttt{REM2})$$
$$\texttt{BINARY} = 1 \rightarrow 1 \rightarrow 0 \rightarrow \texttt{divisible3} \rightarrow \texttt{STOP}$$

We now recall the standard operational semantics of CSP as defined by A.W. Roscoe [24]. It is presented in Figure 3 as a logical inference system. A *state* of the semantics is a process to be evaluated called the *control*. In the following, we assume that the system starts with an initial state `MAIN`, and the rules of the semantics are used to infer how this state evolves. When no rules can be applied to the current state, the computation finishes. The rules of the semantics change the states of the computation due to the occurrence of events. The set of possible events is $\Sigma^\tau = \Sigma \cup \{\tau\}$. Events in Σ are visible from the external environment, and can only happen with its co-operation (e.g., actions of the user). Event τ is an internal event that cannot be observed from outside the system and it happens automatically as defined by the semantics.

(Process Call)	(Prefixing)	(Internal Choice 1)	(Internal Choice 2)
$\dfrac{}{N \xrightarrow{\tau} rhs(N)}$	$\dfrac{}{(a \rightarrow P) \xrightarrow{a} P}$	$\dfrac{}{(P \sqcap Q) \xrightarrow{\tau} P}$	$\dfrac{}{(P \sqcap Q) \xrightarrow{\tau} Q}$
(External Choice 1)	**(External Choice 2)**	**(External Choice 3)**	**(External Choice 4)**
$\dfrac{P \xrightarrow{\tau} P'}{(P \,\Box\, Q) \xrightarrow{\tau} (P' \,\Box\, Q)}$	$\dfrac{Q \xrightarrow{\tau} Q'}{(P \,\Box\, Q) \xrightarrow{\tau} (P \,\Box\, Q')}$	$\dfrac{P \xrightarrow{e} P'}{(P \,\Box\, Q) \xrightarrow{e} P'}\ e \in \Sigma$	$\dfrac{Q \xrightarrow{e} Q'}{(P \,\Box\, Q) \xrightarrow{e} Q'}\ e \in \Sigma$
(Synchronized Parallelism 1)	**(Synchronized Parallelism 2)**	**(Synchronized Parallelism 3)**	
$\dfrac{P \xrightarrow{e} P'}{(P \underset{X}{\parallel} Q) \xrightarrow{e} (P' \underset{X}{\parallel} Q)}\ e \in \Sigma^\tau \backslash X$	$\dfrac{Q \xrightarrow{e} Q'}{(P \underset{X}{\parallel} Q) \xrightarrow{e} (P \underset{X}{\parallel} Q')}\ e \in \Sigma^\tau \backslash X$	$\dfrac{P \xrightarrow{e} P' \quad Q \xrightarrow{e} Q'}{(P \underset{X}{\parallel} Q) \xrightarrow{e} (P' \underset{X}{\parallel} Q')}\ e \in X$	

Figure 3: CSP's operational semantics

In order to perform computations, we begin with the initial state and non-deterministically apply the rules of Figure 3. Their intuitive meaning is the following:

(Process Call) The call to process N is unfolded and $rhs(N)$ becomes the new control.

(Prefixing) When event a occurs, process P becomes the new control.

(Internal Choice 1 and 2) The system, with the occurrence of τ, non-deterministically selects one of the two processes P or Q which becomes the new control.

(External Choice 1, 2, 3 and 4) The occurrence of τ develops one of the processes. The occurrence of an event $e \in \Sigma$ is used to select one of the two processes P or Q and the control changes according to the event.

(Synchronized Parallelism 1 and 2) When a non-synchronized event happens, one of the two processes P or Q evolves accordingly.

(Synchronized Parallelism 3) When a synchronized event ($e \in X$) happens, it is required that both processes synchronize; P and Q are executed at the same time and the control becomes $P' \underset{X}{||} Q'$.

We illustrate the semantics with the following example.

Example 3. *Consider the CSP specification of Example 2. If we execute the semantics, we get the computation shown in Figure 4 where the final state is* $\texttt{STOP} \underset{\{0,1,\texttt{divisible3}\}}{\parallel} \texttt{STOP}$. *This computation corresponds to the case in which the binary number 110, divisible by 3, produces the sequence of events* $\langle 1, 1, 0, \texttt{divisible3}\rangle$. *In the figure, each rewriting step is labeled with the applied rule, and the example should be read top-down.*

Definition 1. *(Traces) Given a process P in a CSP specification, traces(P) is defined as the set of finite sequences of observable events (members of Σ^*). The set of all non-empty, prefix-closed subsets of Σ^* is called the* traces model *(the set of all possible representations of processes using traces).*

For instance, the set of all traces of process `MAIN` in the CSP specification of Example 1 is:

$$traces(\texttt{MAIN}) = \{\langle(\texttt{0}|\texttt{1})^*\rangle, \langle(\texttt{0}^*|\texttt{1}(\texttt{01}^*\texttt{0})^*\texttt{1})^*\texttt{divisible3}\rangle\} \tag{1}$$

The set of all traces of process `MAIN` in the CSP specification of Example 2 is:

$$traces(\texttt{MAIN}) = \{\langle\rangle, \langle\texttt{1}\rangle, \langle\texttt{1},\texttt{1}\rangle, \langle\texttt{1},\texttt{1},\texttt{0}\rangle, \langle\texttt{1},\texttt{1},\texttt{0},\texttt{divisible3}\rangle\} \tag{2}$$

3.2. Labeled Petri nets

In this section, we recall the model of Petri nets by defining some basic concepts needed throughout the paper.

Definition 2. *(Petri Net) A* Petri net *[18, 20] is a tuple $N = (P, T, F)$, where P is a finite set of* places*, T is a finite set of* transitions*, such that $P \cap T = \emptyset$ and $P \cup T \neq \emptyset$ and F is a finite set of weighted* arcs *representing the* flow relation *$F : P \times T \cup T \times P \to \mathbb{N}$. A* marking *of a Petri net is a function $M : P \to \mathbb{N}$. A* marked Petri net *is a pair (N, M_0) where M_0 is a marking of the Petri net called an* initial marking*.*

Definition 3. *(Labeled Petri Net) A* labeled Petri net *[7, 18, 20] is a 6-tuple $\mathcal{N} = (\langle P, T, F\rangle, M_0, \mathcal{P}, \mathcal{T}, \mathcal{L}_P, \mathcal{L}_T)$, where $\langle P, T, F\rangle$ is a Petri net, M_0 is the* initial marking*, $\mathcal{P}$ is a* place alphabet*, $\mathcal{T}$ is a* transition alphabet*, $\mathcal{L}_P$ is a* labelling function *$\mathcal{L}_P : P \to \mathcal{P}$ and $\mathcal{L}_T$ is a* labelling function *$\mathcal{L}_T : T \to \mathcal{T}$.*

In the following, we will use *labeled ordinary Petri nets* where all of its arc weights are 1, $\mathcal{L}_P$ is a partial function and $\mathcal{L}_T$ is a total function. Therefore, because the weight of arcs is always 1, we will consider F as a subset of $P \times T \cup T \times P$. For the sake of concreteness, we often use the notation p_α (t_β) to denote the place p (transition t) whose label is $\alpha \in \mathcal{P}$ ($\beta \in \mathcal{T}$), i.e., $\mathcal{L}_P(p) = \alpha$ ($\mathcal{L}_T(t) = \beta$); or also to assign label α (β) to place p (transition t).

An example of Petri net is drawn in Figure 5. This Petri net is associated with the Moore machine of Example 1 with some extra transitions τ, `C1` and `C2` whose meaning can be ignored for the time being (they will be explained later).

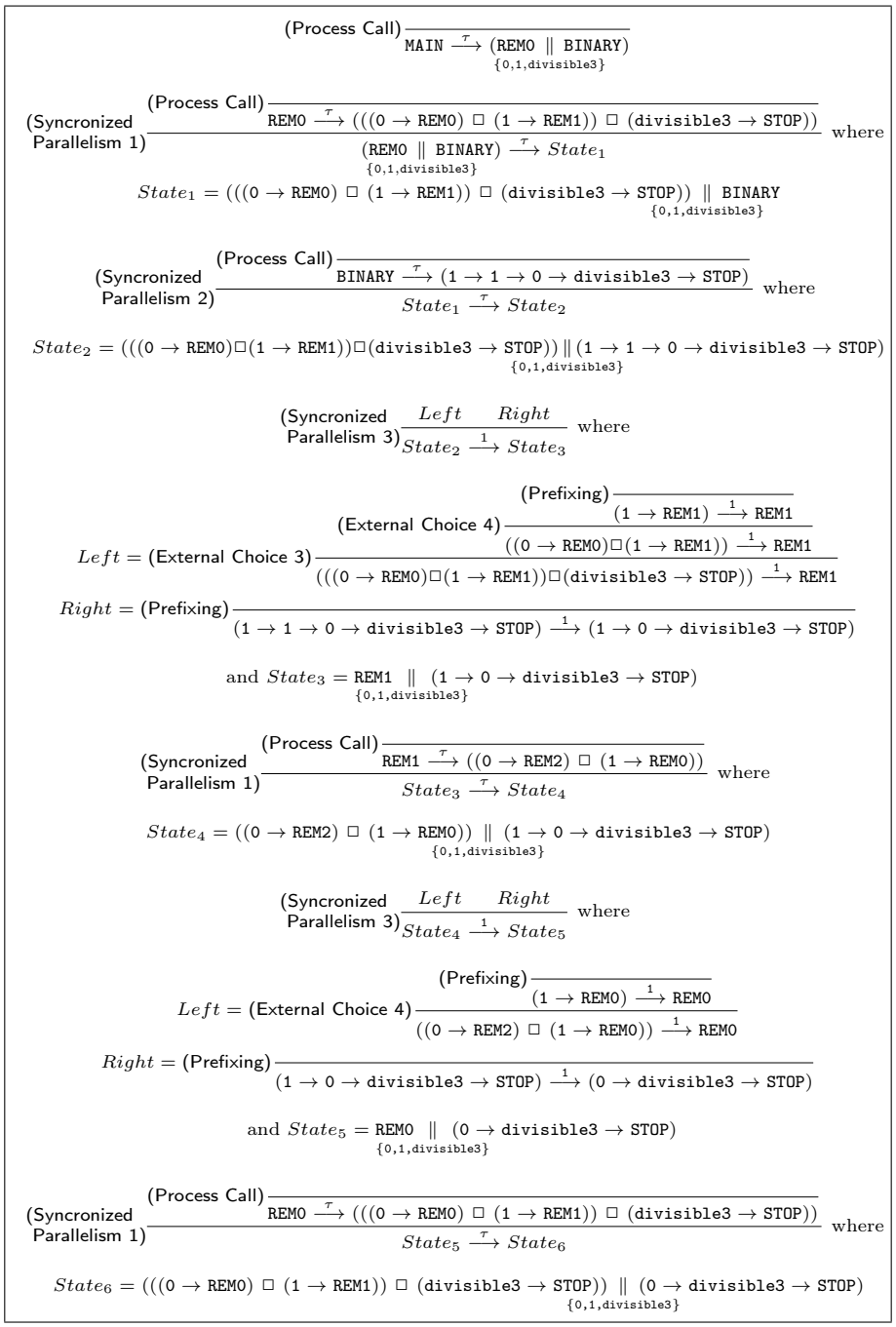


Figure 4: A computation with the operational semantics in Figure 3

Definition 4. *(Input and Output Place/Transition) Given a labeled Petri net* $\mathcal{N} = (\langle P, T, F\rangle, M_0, \mathcal{P}, \mathcal{T}, \mathcal{L}_P, \mathcal{L}_T)$*, we say that a place* $p \in P$ *is an* input (resp. output) place *of a transition* $t \in T$ *if and only if there is an* input (resp. output) arc *from* p *to* t *(resp. from* t *to* p*). Given a transition* $t \in T$*, we denote by* ${}^\bullet t$ *and*

$$\text{(Syncronized Parallelism 3)}\frac{Left \qquad Right}{State_6 \xrightarrow{1} State_7} \text{ where}$$

$$Left = \text{(External Choice 3)}\frac{\text{(Prefixing)}\dfrac{}{(0 \to \texttt{REM0}) \xrightarrow{0} \texttt{REM0}}}{((0 \to \texttt{REM0})\Box(1 \to \texttt{REM1})) \xrightarrow{0} \texttt{REM0}}}{(((0 \to \texttt{REM0})\Box(1 \to \texttt{REM1}))\Box(\texttt{divisible3} \to \texttt{STOP})) \xrightarrow{0} \texttt{REM0}}$$

$$Right = \text{(Prefixing)}\frac{}{(0 \to \texttt{divisible3} \to \texttt{STOP}) \xrightarrow{0} (\texttt{divisible3} \to \texttt{STOP})}$$

$$\text{and } State_7 = \underset{\{0,1,\texttt{divisible3}\}}{\texttt{REM0} \parallel (\texttt{divisible3} \to \texttt{STOP})}$$

$$\text{(Syncronized Parallelism 1)}\frac{\text{(Process Call)}\dfrac{}{\texttt{REM0} \xrightarrow{\tau} (((0 \to \texttt{REM0}) \ \Box \ (1 \to \texttt{REM1})) \ \Box \ (\texttt{divisible3} \to \texttt{STOP}))}}{State_7 \xrightarrow{\tau} State_8} \text{ where}$$

$$State_8 = \underset{\{0,1,\texttt{divisible3}\}}{(((0 \to \texttt{REM0}) \ \Box \ (1 \to \texttt{REM1})) \ \Box \ (\texttt{divisible3} \to \texttt{STOP})) \parallel (\texttt{divisible3} \to \texttt{STOP})}$$

$$\text{(Syncronized Parallelism 3)}\frac{Left \qquad Right}{State_8 \xrightarrow{\texttt{divisible3}} \underset{\{0,1,\texttt{divisible3}\}}{\texttt{STOP} \parallel \texttt{STOP}}} \text{ where}$$

$$Left = \text{(External Choice 4)}\frac{\text{(Prefixing)}\dfrac{}{(\texttt{divisible3} \to \texttt{STOP}) \xrightarrow{\texttt{divisible3}} \texttt{STOP}}}{(((0 \to \texttt{REM0})\Box(1 \to \texttt{REM1}))\Box(\texttt{divisible3} \to \texttt{STOP})) \xrightarrow{\texttt{divisible3}} \texttt{STOP}}$$

$$Right = \text{(Prefixing)}\frac{}{(\texttt{divisible3} \to \texttt{STOP}) \xrightarrow{\texttt{divisible3}} \texttt{STOP}}$$

Figure 4: A computation with the operational semantics in Figure 3 (cont.)

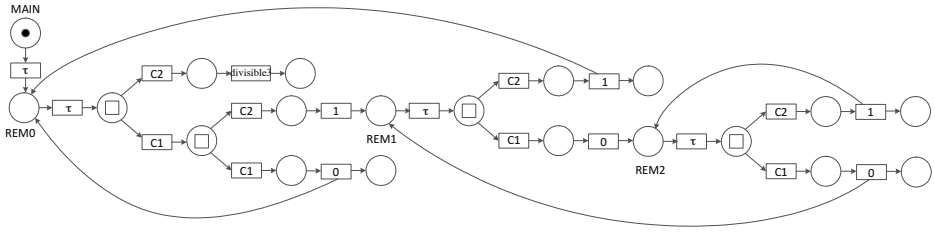


Figure 5: Petri net associated with the specification of Example 1

$t^\bullet$ *the set of all input and output places of* t*, respectively. Analogously, given a place* $p \in P$*, we denote* ${}^\bullet p$ *and* $p^\bullet$ *the set of all input and output transitions of* p*, respectively. Formally,*

$$ {}^\bullet t = \{p \in P \mid (p,t) \in F\} \text{ and } t^\bullet = \{p \in P \mid (t,p) \in F\}$$
$$ {}^\bullet p = \{t \in T \mid (t,p) \in F\} \text{ and } p^\bullet = \{t \in T \mid (p,t) \in F\}$$

The notion of firing sequence is used in the paper according to the generally accepted definition [7, 18, 20].

Definition 5. *(Enabling and Firing a Transition) Given a labeled Petri net* $\mathcal{N} = (\langle P, T, F\rangle, M_0, \mathcal{P}, \mathcal{T}, \mathcal{L}_P, \mathcal{L}_T)$*, we say that a transition* $t \in T$ *is* enabled *in marking* M*, in symbols* $M \xrightarrow{t}$*, if and only if for each input place* $p \in {}^\bullet t$*, we have* $M(p) \geq 1$*. A transition may only be fired if it is enabled.*

The firing *of an enabled transition t in a marking M eliminates one token from each input place $p \in {}^\bullet t$ and adds one token to each output place $p' \in t^\bullet$, producing a new marking M', in symbols $M \xrightarrow{t} M'$. Formally,*

$$M'(p) = \begin{cases} M(p) - 1 & \text{if } p \in {}^\bullet t \wedge p \notin t^\bullet \\ M(p) + 1 & \text{if } p \notin {}^\bullet t \wedge p \in t^\bullet \\ M(p) & \text{otherwise} \end{cases}$$

We say that a marking M_n is reachable *from an initial marking M_0 if there is a* firing sequence *$\sigma = t_1 t_2 \dots t_n$ such that $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \dots \xrightarrow{t_n} M_n$. In this case, we say that M_n is reachable from M_0 through σ, in symbols $M_0 \xrightarrow{\sigma} M_n$. This notion includes the empty sequence ϵ; we have $M \xrightarrow{\epsilon} M$ for any marking M. The set of all reachable markings from M_0 is denoted by $R(M_0)$.*

Definition 6. *(Petri Net Language) Given a labeled Petri net $\mathcal{N} = (\langle P, T, F\rangle, M_0, \mathcal{P}, \mathcal{T}, \mathcal{L}_P, \mathcal{L}_T)$, we denote by $FS(\mathcal{N})$ the set of all possible firing sequences over T:*

$$FS(\mathcal{N}) = \bigcup_{M \in R(M_0)} \{\sigma \mid M_0 \xrightarrow{\sigma} M\}$$

The language *$L_\mathcal{A}(\mathcal{N})$ (over the alphabet $\mathcal{A}$) is given by:*

$$\begin{aligned} L_\mathcal{A}(\mathcal{N}) = \{\langle\rangle\} \cup \{\langle \mathcal{L}_T(s_1), \dots, \mathcal{L}_T(s_m)\rangle \mid \exists \sigma = t_1 \dots t_n \in FS(\mathcal{N}) \wedge \\ \forall i, 1 \le i \le m \text{ with } m \le n : s_i = t_k \text{ with } 1 \le k \le n \wedge \\ \mathcal{L}_T(t_k) \in \mathcal{A} \wedge \nexists j, 1 \le j < i : s_j = t_l \text{ with } l \ge k \ \} \end{aligned}$$

Whenever $\mathcal{A}$ is not specified it is assumed that $\mathcal{A} = \mathcal{T}$.

For instance, the (infinite) language produced by the labeled Petri net in Figure 5 is:

$$\begin{aligned} L(\mathcal{N}) = \{&\langle\rangle, \langle\tau\rangle, \langle\tau,\ \tau\rangle, \langle\tau,\ \tau,\ \texttt{C2}\rangle, \langle\tau,\ \tau,\ \texttt{C2},\ \texttt{divisible3}\rangle, \langle\tau,\ \tau,\ \texttt{C1}\rangle, \\ &\langle\tau,\ \tau,\ \texttt{C1},\ \texttt{C1}\rangle, \langle\tau,\ \tau,\ \texttt{C1},\ \texttt{C1},\ \texttt{0}\rangle, \langle\tau,\ \tau,\ \texttt{C1},\ \texttt{C2}\rangle, \langle\tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ \texttt{1}\rangle, \\ &\langle\tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ \texttt{1},\ \tau\rangle, \langle\tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ \texttt{1},\ \tau,\ \texttt{C2}\rangle, \\ &\langle\tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ \texttt{1},\ \tau,\ \texttt{C2},\ \texttt{1}\rangle, \dots\} \end{aligned}$$

4. Transformation of a CSP specification into an equivalent Petri net

This section introduces an algorithm to transform a CSP specification into an equivalent Petri net. We first provide a notion of equivalence which is based on the traces generated by the initial CSP and the language produced by the final Petri net, and that allows us to formally prove the correctness of the transformation.

In particular, given a CSP specification, the Petri net generated by our algorithm is equivalent to the CSP in the sense that the sequences of observable events produced are exactly the same in both models (i.e., they are equivalent modulo a given alphabet). In CSP terminology, these sequences are the so-called traces (see, e.g., chapter 8.2 of [24]). In Petri nets they correspond to transition firing sequences (see, e.g., [18]). Formally,

Definition 7. *(Equivalence between CSP and Petri nets) Given a CSP specification $\mathcal{S}$ and a Petri net $\mathcal{N}$, we say that $\mathcal{S}$ is* equivalent *to $\mathcal{N}$ if and only if $traces(\texttt{MAIN}) = L_\Sigma(\mathcal{N})$, where process* `MAIN` *belongs to $\mathcal{S}$.*

Observe that the Petri net produces a language module the alphabet Σ which contains all the external events of $\mathcal{S}$. Note also that $traces(\texttt{MAIN})$ contains only events that are external (i.e., observable from outside the system). Therefore, this notion of equivalence implies that, if we ignore internal events such as τ, then the sequences of (observable) actions of both systems are exactly the same.

Algorithm 1 General Algorithm

Input: A CSP specification $\mathcal{S}$ with initial process `MAIN`
Output: A labeled Petri net $\mathcal{N}$ equivalent to $\mathcal{S}$

Build the initial state of the semantics: $state = (\texttt{MAIN}, p_0, \mathcal{N}_0, ([], []), \emptyset)$
where $\mathcal{N}_0 = (\langle\{p_0\}, \emptyset, \emptyset\rangle, M_0, \mathcal{P}, \mathcal{T}, \mathcal{L}_P, \mathcal{L}_T)$, $M_0(p_0) = 1$,
$\mathcal{P} = Names \cup \{\square, \sqcap\}$, $\mathcal{T} = \Sigma^\tau \cup \{\|, \texttt{C1}, \texttt{C2}\}$.
repeat
  repeat
    Run the rules of the instrumented semantics with the state *state*
  until no more rules can be applied
  Get the new state $state = (\_, \_, \mathcal{N}, ([], S), \_)$
  $state = (\texttt{MAIN}, p_0, \mathcal{N}, (\texttt{UpdStore}(S), []), \emptyset)$
until $\texttt{UpdStore}(S) = []$
return $\mathcal{N}$
where function `UpdStore` is defined as follows:

$$\texttt{UpdStore}(S) = \begin{cases} (rule, rules \backslash \{rule\}) : S' & \text{if } S = (\_, rules) : S' \wedge rule \in rules \\ \texttt{UpdStore}(S') & \text{if } S = (\_, \emptyset) : S' \\ [] & \text{if } S = [] \end{cases}$$

Even though the transformation is controlled by an algorithm, the generation of the final Petri net is carried out by an instrumented operational semantics of CSP. In particular, the algorithm fires the execution of the semantics that generates incrementally and as a side effect the Petri net.

The instrumentation of the semantics performs three main tasks:

1. It produces a computation and generates as a side-effect a Petri net associated with the computation.
2. It controls that no infinite loops are executed.
3. It ensures that the execution is deterministic.

The transformation is directed by Algorithm 1. This algorithm controls the execution of the semantics and repeatedly uses it to deterministically execute all possible computations—of the original (non-deterministic) specification—and the Petri net is constructed incrementally with each execution of the semantics. Concretely, each time the semantics is executed, it produces as a result a portion of the Petri net. This result is the input of the next execution of the semantics that adds a new part of the Petri net. This process is repeated until

all possible executions have been explored and thus the complete Petri net has been produced.

The key point of the algorithm is the use of a store that records the actions that can be performed by the semantics. In particular, the store is an ordered list of elements that allows us to add and extract elements from the beginning and the end of the list; and it contains tuples of the form $(rule, rules)$ where:

- *rule* indicates the rule that must be selected by the semantics in the next execution step, when different possibilities exist. They are indicated with one of the following intuitive abreviatures `SP1`, `SP2`, `SP3`, `SP4`, `C1` and `C2`. Thanks to *rule* the semantics is deterministic because it knows at every step what rule must be applied.

- *rules* is a set containing the other possible rules that can be selected with the current control. Therefore, *rules* records at every step all the possible rules not applied so that the algorithm will execute the semantics again with these rules.

The algorithm uses the store to prepare each execution of the semantics indicating the rules that must be applied at each step. For this, function `UpdStore` is used; it basically avoids to repeat the same computation with the semantics. When the semantics finishes, the algorithm prepares a new execution of the semantics with an updated store. This is repeated until all possible computations are explored (i.e., until the store is empty).

Taking into account that the semantics in Figure 3 can be non-terminating (it can produce infinite computations), the instrumented semantics could be also non-terminating if a loop-checking mechanism is not incorporated to ensure termination. In order to ensure termination of all computations, the instrumentation of the semantics incorporates a mechanism to stop the computation when the same process is repeated in the same context (i.e., the same control appears twice in a (sub)derivation of the semantics).

The instrumented semantics used by Algorithm 1 is shown in Figure 6. It is an operational semantics where a *state* is a tuple $(P, p, \mathcal{N}, (S, S_0), \Delta)$, where:

- P is the process to be evaluated (the *control*),

- p is the last place added to the Petri net $\mathcal{N}$,

- (S, S_0) is a tuple with two stores (where the empty store is denoted by $[]$) that contains the rules to apply and the rules applied so far, and

- Δ is a set of references used to insert synchronizations in $\mathcal{N}$.

The basic idea of the Petri net construction is to generate the Petri net associated with the current control and connect this net to the last place added to $\mathcal{N}$.

Given a labeled Petri net $\mathcal{N} = (\langle P, T, F\rangle, M, \mathcal{P}, \mathcal{T}, \mathcal{L}_P, \mathcal{L}_T)$ and the current reference $p \in P$, we use the notation $\mathcal{N}[p \mapsto t_a \mapsto p']$ either as a condition on $\mathcal{N}$ (i.e., $\mathcal{N}$ contains transition t_a), or also to introduce a transition t and a place p' into $\mathcal{N}$ producing the net $\mathcal{N}' = (\langle P', T', F'\rangle, M', \mathcal{P}, \mathcal{T}, \mathcal{L}_P, \mathcal{L}_T)$ where $P' = P \cup \{p'\}$, $T' = T \cup \{t_a\}$, $F' = F \cup \{(p, t_a), (t_a, p')\}$, $\forall p \in P : M'(p) = M(p)$, $M'(p') = 0$ and $\mathcal{L}_T(t_a) = a$ where $a \in \mathcal{T}$.

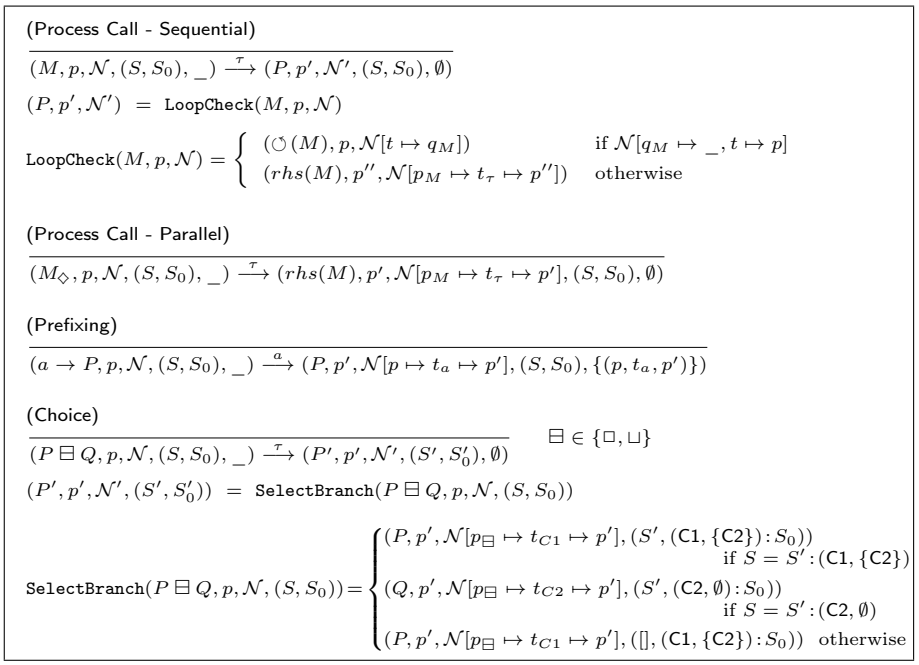


Figure 6: An instrumented operational semantics that generates a Petri net

An explanation for each rule of the semantics follows:

(Process Call - Sequential) In the instrumented semantics, there are two versions of the standard rule for process call. The first version is used when a process call is made in a sequential process. The second version is used for process calls made inside parallelism operators. The sequential version basically decides whether process P must be unfolded or not. This is done to avoid infinite unfolding of the same process. Once a (sequential) process has been unfolded once, it is not unfolded again. This is controlled with function `LoopCheck`. If the process has been previously unfolded (thus, a place q with the label M already belongs to the Petri net $\mathcal{N}$, i.e., $\mathcal{N}[q_M \mapsto \_]$), then we are in a loop, and P is marked as a loop with the special symbol $\circlearrowleft$. This label avoids to unfold the process again because no rule is applicable. In this case, to represent the loop in the Petri net, we add a new arc from the last added transition to the place q_M ($\mathcal{N}[t \mapsto q_M]$). If P has not been previously unfolded, then $rhs(P)$ becomes the new control. Observe that the new Petri net $\mathcal{N}'$ contains a place p_M that represents the process call and a transition t_τ that represents the occurrence of event τ. No event in Σ is fired in this rule, thus no synchronization is possible and Δ is empty.

(Process Call - Parallel) When a process call is made inside a parallelism operator, it is always unfolded. We do not worry about infinite unfolding because the rules for synchronized parallelism already control non-termination. In order to distinguish between process calls made sequentially or in parallel, we use a special symbol $\diamondsuit$. Therefore, for simplicity, we assume that all process calls inside parallelisms are labeled with $\diamondsuit$, and thus, the semantics can decide what rule should be used.

(Prefixing) This rule adds to $\mathcal{N}$ a transition t_a that represents the occurrence of event a. t_a is connected to the current place p and to a new place p'. The new control is P. The new set Δ contains the tuple (p, t_a, p') to indicate that event a must be synchronized when required by (Synchronized Parallelism 3).

(Choice) The only sources of non-determinism are choice operators (different branches can be selected for execution) and parallel operators (different order of branches can be selected for execution). Therefore, every time the semantics executes a choice or a parallelism, they are made deterministic thanks to the information in the store S. In the case of choices, both internal and external can be treated with a single rule. We use symbol $\boxminus$ to refer to both $\{\Box, \sqcup\}$. In this rule, function `SelectBranch` is used to produce the new control P' and the new tuple of stores (S', S'_0), by selecting a branch with the information of the store. Note that, for simplicity, the lists constructor ":" has been overloaded, and it is also used to build lists of the form $(A : a)$ where A is a list and a is the last element.

If the last element of the store S indicates that the first branch of the choice (C1) must be selected, then P is the new control. If the second branch must be selected (C2), the new control is Q. In any other case the store is empty, and thus this is the first time that this choice is evaluated. Then, we select the first branch (P is the new control) and we add $(\mathsf{C1}, \{\mathsf{C2}\})$ to the store S_0 indicating that C1 has been chosen, and the remaining option is C2. This function creates a new transition for each branch (t_{C1} and t_{C2}) that represents the τ event.

(Synchronized Parallelism 1 and 2) The store determines what rule to use when a parallelism operator is in the control. If we are not in a loop (this is known because the same control has not appeared before, i.e., $(P1 \| P2, \_, \_) \notin \Upsilon$) and the last element in the store is SP1, then (Synchronized Parallelism 1) is used. If it is SP2, (Synchronized Parallelism 2) is used. In a synchronized parallelism composition, both parallel processes can be intertwiningly executed until a synchronized event is found. Therefore, places and transitions for both processes can be added interwoven to the Petri net. Hence, the semantics needs to know in every state the references to be used in both branches. This is done by labeling each parallelism operator with a tuple of the form (p_1, p_2, Υ) where p_1 and p_2 are respectively the last places added to the left and right branches of the parallelism; and Υ records the controls of the semantics in order to avoid repetition (i.e., it is used to avoid infinite loops). In particular, Υ is a set of triples of the form: $(P1 \| P2, p_1, p_2)$ where $P1 \| P2$ is the control of a previous state of the semantics, and p_1, p_2 are the nodes in the Petri net associated with $P1$ and $P2$. This tuple is initialized to $(\bot, \bot, \emptyset)$ for every parallelism that is introduced in the computation. Here, we use symbol $\bot$ to denote an undefined place. The new label of the parallelism contains a new Υ that has been updated with the current control only if it does not contain any $\circlearrowleft$. This is done with a simple syntactic checking performed with function `HasLoops`. The set Δ is passed down unchanged so that rule (Synchronized Parallelism 3) can use it if necessary.

These rules develop the branches of the parallelism until they are finished or until they must synchronize. They use function `InitBranches` to introduce the parallelism into the Petri net the first time it is executed. Observe that the parallelism operator is represented in the Petri net with a transition $t_{\|}$. This

(Synchronized Parallelism 1)

$$\frac{(P1, p_1', \mathcal{N}', (S', (\mathsf{SP1}, rules) : S_0), \_) \xrightarrow{e} (P1', p_1'', \mathcal{N}'', (S'', S_0'), \Delta)}{(P1 \underset{X}{\parallel_{lab}} P2, p, \mathcal{N}, (S' : (\mathsf{SP1}, rules), S_0), \_) \xrightarrow{e} (P1' \underset{X}{\parallel_{lab'}} P2, p, \mathcal{N}'', (S'', S_0'), \Delta)} \quad e \in \Sigma^\tau \backslash X$$

$lab = (p_1, p_2, \Upsilon) \ \wedge \ (P1 \| P2, \_, \_) \notin \Upsilon \ \wedge$

$(\mathcal{N}', p_1', p_2') \ = \ \texttt{InitBranches}(\mathcal{N}, p_1, p_2, p) \ \wedge \ lab' = (p_1'', p_2', \texttt{NewUpsilon}(\Upsilon, (P1 \| P2, p_1', p_2')))$

(Synchronized Parallelism 2)

$$\frac{(P2, p_2', \mathcal{N}', (S', (\mathsf{SP2}, rules) : S_0), \_) \xrightarrow{e} (P2', p_2'', \mathcal{N}'', (S'', S_0'), \Delta)}{(P1 \underset{X}{\parallel_{lab}} P2, p, \mathcal{N}, (S' : (\mathsf{SP2}, rules), S_0), \_) \xrightarrow{e} (P1 \underset{X}{\parallel_{lab'}} P2', p, \mathcal{N}'', (S'', S_0'), \Delta)} \quad e \in \Sigma^\tau \backslash X$$

$lab = (p_1, p_2, \Upsilon) \ \wedge \ (P1 \| P2, \_, \_) \notin \Upsilon \ \wedge$

$(\mathcal{N}', p_1', p_2') \ = \ \texttt{InitBranches}(\mathcal{N}, p_1, p_2, p) \ \wedge \ lab' = (p_1', p_2'', \texttt{NewUpsilon}(\Upsilon, (P1 \| P2, p_1', p_2')))$

(Synchronized Parallelism 3)

$$\frac{Left \quad Right}{(P1 \underset{X}{\parallel_{lab}} P2, p, \mathcal{N}, (S' : (\mathsf{SP3}, rules), S_0), \_) \xrightarrow{e} (P1' \underset{X}{\parallel_{lab'}} P2', p, \mathcal{N}_s, (S''', S_0''), \Delta)} \quad e \in X$$

$lab = (p_1, p_2, \Upsilon) \ \wedge \ (P1 \| P2, \_, \_) \notin \Upsilon \ \wedge \ (\mathcal{N}', p_1', p_2') \ = \ \texttt{InitBranches}(\mathcal{N}, p_1, p_2, p) \ \wedge$

$Left = (P1, p_1', \mathcal{N}', (S', (\mathsf{SP3}, rules) : S_0), \_) \xrightarrow{e} (P1', p_1'', \mathcal{N}'', (S'', S_0'), \Delta_1) \ \wedge$

$Right = (P2, p_2', \mathcal{N}'', (S'', S_0'), \_) \xrightarrow{e} (P2', p_2'', \mathcal{N}''', (S''', S_0''), \Delta_2) \ \wedge$

$lab' = (p_1'', p_2'', \texttt{NewUpsilon}(\Upsilon, (P1 \| P2, p_1', p_2'))) \ \wedge$

$\mathcal{N}_s = (\mathcal{N}''' \cup \{(p \mapsto t_e \mapsto p') \mid (p, \_, p') \in (\Delta_1 \cup \Delta_2)\}) \backslash \{(p \mapsto t \mapsto p') \mid (p, t, p') \in (\Delta_1 \cup \Delta_2)\}$

$\wedge \ \Delta = \{(p, t_e, p') \mid (p, \_, p') \in (\Delta_1 \cup \Delta_2)\}$

$$\texttt{InitBranches}(\mathcal{N}, p_1, p_2, p) = \begin{cases} (\mathcal{N}[p \mapsto t_\| \mapsto p_1', p \mapsto t_\| \mapsto p_2'], p_1', p_2') & \text{if } p_1 = \bot \\ (\mathcal{N}, p_1, p_2) & \text{otherwise} \end{cases}$$

$$\texttt{NewUpsilon}(\Upsilon, (P1 \| P2, p_1, p_2)) = \begin{cases} \Upsilon & \text{if } \texttt{HasLoops}(P1 \| P2) \\ \Upsilon \cup \{(P1 \| P2, p_1, p_2)\} & \text{otherwise} \end{cases}$$

Figure 6: An instrumented operational semantics that generates a Petri net (cont.)

transition is connected to two new places (p_1' and p_2'), one for each branch. After executing function `InitBranches`, we get a new net and new references for each branch.

(Synchronized Parallelism 3) It is applied when the last element in the store is SP3 and no loop is detected. It is used to synchronize the parallel processes. In this rule, all the events that have been executed in this step must be synchronized. Therefore, all the events occurred in the subderivations of $P1$ (Δ_1) and $P2$ (Δ_2) are mutually synchronized. Note that this is done in the Petri net by removing the transitions that were added in each subderivation ($\{(p \mapsto t \mapsto p') \mid (p, t, p') \in (\Delta_1 \cup \Delta_2)\}$) and connecting all of them with a single transition t_e, $e \in X$. The new Δ contains all the synchronizations occurred in both branches connected by the new transition t_e ($\Delta = \{(p, t_e, p') \mid (p, \_, p') \in (\Delta_1 \cup \Delta_2)\}$).

(Synchronized Parallelism 4) This rule is applied when the last element in the store is SP4. It is used when none of the parallel processes can proceed (because they already finished, deadlocked or were labeled with $\circlearrowleft$). When a parallelism

is labeled as a loop with ↻, it can be unlabeled to unfold it once[1] in order to allow the other processes to continue. This happens when the looped process is in parallel with other process and the later is waiting to synchronize with the former. In order to perform the synchronization, both processes must continue, thus the loop is unlabeled. This task is done by function `LoopControl`. It decides whether the branches of the parallelism should be further unfolded or they should be stopped (e.g., due to a deadlock or an infinite loop). `LoopControl` can detect three different situations:

(i) The parallelism is in a loop. In this case, the whole parallelism is marked as a loop. This situation happens when one of the branches is marked as looped (with ↻), and the other branch is also looped, or it already terminated (i.e., it is `STOP`), or the control of both branches of the parallelism have been repeated (i.e., they are in Υ).

(ii) The parallelism is not in a loop, and it should proceed. This situation happens when one of the branches is marked as looped, and the other branch is trying to synchronize with the first one. In this case, the branch marked as a loop should continue to allow the synchronization. Therefore, the loop symbol ↻ is removed and the loop arcs added to the Petri net $\mathcal{N}$ are also recursively removed with function `DelEdges`.

(iii) The parallelism must be stopped. This happens for instance because both branches terminated, therefore, the whole parallelism is replaced by `STOP`, thus, stopping further computations.

(Synchronized Parallelism 4)

$$\frac{}{(P1 \underset{X}{\|}_{(p_1,p_2,\Upsilon)} P2, p, \mathcal{N}, (S' : (\mathsf{SP4}, rules), S_0), \_) \xrightarrow{\tau} (P', p, \mathcal{N}', (S', (\mathsf{SP4}, rules) : S_0), \emptyset)}$$

$$(P', \mathcal{N}') = \texttt{LoopControl}(P1 \underset{X}{\|}_{(p_1,p_2,\Upsilon)} P2, \mathcal{N})$$

$$\texttt{LoopControl}(P1 \underset{X}{\|}_{(p_1,p_2,\Upsilon)} P2, \mathcal{N}) =$$

$$\begin{cases} (\circlearrowleft(P1'' \underset{X}{\|}_{(p_1',p_2',\Upsilon)} P2''), \mathcal{N}) & \text{if } P1' = \circlearrowleft(P1'') \wedge (P2' = \circlearrowleft(P2'') \vee \\ & \quad ((P2' = \texttt{STOP} \vee (P1'' \| P2', \_, \_) \in \Upsilon) \ \wedge \ P2'' = P2')) \\ (P1''' \underset{X}{\|}_{(p_1',p_2',\Upsilon)} P2', \mathcal{N}') & \text{if } P1' = \circlearrowleft(P1'') \wedge P2' \neq \circlearrowleft(\_) \wedge P2' \neq \texttt{STOP} \wedge \\ & \quad (P1'' \| P2', \_, \_) \notin \Upsilon \ \wedge \ (P1''', \mathcal{N}') = \texttt{DelEdges}(P1'', \mathcal{N}) \\ (\texttt{STOP}, \mathcal{N}) & \text{otherwise} \end{cases}$$

where $(P1', p_1', P2', p_2') \in \{(P1, p_1, P2, p_2), (P2, p_2, P1, p_1)\}$

$$\texttt{DelEdges}(P1 \underset{X}{\|}_{(p_1,p_2,\Upsilon)} P2, \mathcal{N}) =$$

$$\begin{cases} (P1' \underset{X}{\|}_{(p_1,p_2,\Upsilon')} P2', \mathcal{N}'') & \text{if } (P1 \| P2, pp_1, pp_2) \in \Upsilon \wedge \Upsilon' = \Upsilon \backslash \{(P1 \| P2, pp_1, pp_2)\} \wedge \\ & \quad ((P1 \neq (\_\|\_) \wedge \mathcal{N}' = \mathcal{N}[t_1 \mapsto p_1] \backslash \{t_1 \mapsto pp1\} \wedge P1' = P1) \\ & \quad \vee (P1 = (\_\|\_) \wedge (P1', \mathcal{N}') = \texttt{DelEdges}(P1, \mathcal{N}))) \wedge \\ & \quad ((P2 \neq (\_\|\_) \wedge \mathcal{N}'' = \mathcal{N}'[t_2 \mapsto p_2] \backslash \{t_2 \mapsto pp2\} \wedge P2' = P2) \\ & \quad \vee (P1 = (\_\|\_) \wedge (P2', \mathcal{N}'') = \texttt{DelEdges}(P2, \mathcal{N}'))) \\ (P1 \underset{X}{\|}_{(p_1,p_2,\Upsilon)} P2, \mathcal{N}) & \text{otherwise} \end{cases}$$

Figure 6: An instrumented operational semantics that generates a Petri net (cont.)

(Synchronized Parallelism 5) This rule is used to detect loops (when the control has been repeated and thus it appears in Υ, and SP1, SP2 or SP3 is the last

[1] Only once because it will be labeled again when the loop is repeated.

element in the store), and also to determine what rule must be applied (when the store is empty).

In order to control non-termination, this rule uses function `CheckLoops` to check whether the current control or other parallelisms inside it has been already repeated in the computation (this is done with the information in Υ). If this is the case, then we are in a loop, and the parallelisms are labeled with the symbol

(Synchronized Parallelism 5)

$$\frac{(P, p, \mathcal{N}_P, (S'_P, S_0), \_) \xrightarrow{e} (P', p, \mathcal{N}', (S', S'_0), \Delta)}{(P1 \underset{X}{\|}_{(p_1,p_2,\Upsilon)} P2, p, \mathcal{N}, (S, S_0), \_) \xrightarrow{e} (P'', p, \mathcal{N}'', (S'', S''_0), \Delta')} \quad e \in \Sigma^\tau$$

$$S = [] \ \vee \ ((S = (\_ : (\mathsf{SP1}, \_)) \ \vee \ S = (\_ : (\mathsf{SP2}, \_)) \ \vee \ S = (\_ : (\mathsf{SP3}, \_))) \\ \wedge (P1 \underset{X}{\|} P2, \_, \_) \in \Upsilon)$$

$$\wedge (P, \mathcal{N}_P, S_P) = \texttt{CheckLoops}(P1 \underset{X}{\|}_{(p1,p2,\Upsilon)} P2, \mathcal{N})$$

$$\wedge((S = [] \wedge S_P = [] \wedge e = \tau \wedge (P'', \mathcal{N}'', (S'', S''_0), \Delta') = (P, \mathcal{N}_P, ([], S_0), \emptyset)) \\ \vee ((S = [] \wedge S_P \neq [] \wedge S'_P = S_P) \vee (S \neq [] \wedge S'_P = S) \\ \wedge (P'', \mathcal{N}'', (S'', S''_0), \Delta') = (P', \mathcal{N}', (S', S'_0), \Delta)))$$

$$\texttt{CheckLoops}(P1 \underset{X}{\|}_{(p1,p2,\Upsilon)} P2, \mathcal{N}) =$$

$$\begin{cases} (\circlearrowleft(P1 \underset{X}{\|}_{(p1,p2,\Upsilon)} P2), \mathcal{N}''', []) & \text{if } (P1 \| P2, pp_1, pp_2) \in \Upsilon \\ & \quad \wedge \ ((\mathcal{N}'' = \mathcal{N}[t_1 \mapsto p_1, t_1 \mapsto pp_1] \ \wedge \ P1 \neq (\_\|\_)) \\ & \qquad \vee \ (\mathcal{N}'' = \mathcal{N} \wedge P1 = (\_\|\_))) \\ & \quad \wedge \ ((\mathcal{N}''' = \mathcal{N}''[t_2 \mapsto p_2, t_2 \mapsto pp_2] \ \wedge \ P2 \neq (\_\|\_)) \\ & \qquad \vee \ (\mathcal{N}''' = \mathcal{N}'' \wedge P2 = (\_\|\_))) \\ ((P1 \underset{X}{\|}_{(p1,p2,\Upsilon)} P2), \mathcal{N}_1 \cup \mathcal{N}_2, S') & \text{otherwise} \end{cases}$$

$$\text{where } (P1', \mathcal{N}_1, S_1) = \begin{cases} \texttt{CheckLoops}(P1, \mathcal{N}) & \text{if } P1 = \_\|\_ \\ (P1, \mathcal{N}, []) & \text{otherwise} \end{cases}$$

$$(P2', \mathcal{N}_2, S_2) = \begin{cases} \texttt{CheckLoops}(P2, \mathcal{N}) & \text{if } P2 = \_\|\_ \\ (P2, \mathcal{N}, []) & \text{otherwise} \end{cases}$$

$$Rules = \texttt{AppRules}(P1' \underset{X}{\|} P2')$$

$$S' = \begin{cases} S_2 : (\{\mathsf{SP2}\}, \emptyset) & \text{if } P1 = \_\|\_ \wedge P2 \neq \_\|\_ \wedge S_1 = [] \wedge Rules = \{\mathsf{SP2}\} \\ S_1 : (\{\mathsf{SP1}\}, \emptyset) & \text{if } P1 \neq \_\|\_ \wedge P2 = \_\|\_ \wedge S_2 = [] \wedge Rules = \{\mathsf{SP1}\} \\ S' : (r, Rules \backslash \{r\}) & \text{if } P1 \neq \_\|\_ \wedge P2 \neq \_\|\_ \wedge \mathsf{SP4} \notin Rules \\ & \quad \wedge r \in Rules \wedge (\ (S' = S_1 \wedge r = \mathsf{SP1}) \\ & \qquad \vee (S' = S_2 \wedge r = \mathsf{SP2}) \\ & \qquad \vee (S' = S_2 \cdot S_1 \wedge r = \mathsf{SP3})\ ) \\ [(\{\mathsf{SP4}\}, \emptyset)] & \text{otherwise} \end{cases}$$

$$\texttt{AppRules}(P1 \underset{X}{\|} P2) = \begin{cases} \{\mathsf{SP1}\} & \text{if } \tau \in \texttt{FstEvs}(P1) \\ \{\mathsf{SP2}\} & \text{if } \tau \notin \texttt{FstEvs}(P1) \wedge \tau \in \texttt{FstEvs}(P2) \\ R & \text{if } \tau \notin \texttt{FstEvs}(P1) \wedge \tau \notin \texttt{FstEvs}(P2) \ \wedge \ R \neq \emptyset \\ \{\mathsf{SP4}\} & \text{otherwise} \end{cases}$$

$$\text{where } \begin{cases} \mathsf{SP1} \in R & \text{if } \exists e \in \texttt{FstEvs}(P1) \wedge e \notin X \\ \mathsf{SP2} \in R & \text{if } \exists e \in \texttt{FstEvs}(P2) \wedge e \notin X \\ \mathsf{SP3} \in R & \text{if } \exists e \in \texttt{FstEvs}(P1) \wedge \exists e \in \texttt{FstEvs}(P2) \wedge e \in X \end{cases}$$

$$\texttt{FstEvs}(P) = \begin{cases} \{a\} & \text{if } P = a \to Q \\ \emptyset & \text{if } P = \circlearrowleft Q \ \vee \ P = \texttt{STOP} \\ \{\tau\} & \text{if } P = M \ \vee \ P = Q \,\square\, R \ \vee \ P = (\texttt{STOP} \underset{X}{\|} \texttt{STOP}) \\ & \quad \vee \ P = (\circlearrowleft Q \underset{X}{\|} \circlearrowleft R) \ \vee \ P = (\circlearrowleft Q \underset{X}{\|} \texttt{STOP}) \ \vee \ P = (\texttt{STOP} \underset{X}{\|} \circlearrowleft R) \\ & \quad \vee (P = (\circlearrowleft Q \underset{X}{\|} R) \wedge \texttt{FstEvs}(R) \subseteq X) \vee (P = (Q \underset{X}{\|} \circlearrowleft R) \wedge \texttt{FstEvs}(Q) \subseteq X) \\ & \quad \vee (P = Q \underset{X}{\|} R \wedge \texttt{FstEvs}(Q) \subseteq X \wedge \texttt{FstEvs}(R) \subseteq X \wedge \bigcap_{M \in \{Q,R\}} \texttt{FstEvs}(M) = \emptyset) \\ E & \text{otherwise, with } P = Q \underset{X}{\|} R \wedge E = (\texttt{FstEvs}(Q) \cup \texttt{FstEvs}(R)) \backslash \\ & \qquad (X \cap (\texttt{FstEvs}(Q) \backslash \texttt{FstEvs}(R) \cup \texttt{FstEvs}(R) \backslash \texttt{FstEvs}(Q))) \end{cases}$$

Figure 6: An instrumented operational semantics that generates a Petri net (cont.)

$\circlearrowleft$; thus it cannot continue unless this symbol is removed by other parallel process that requires the unfolding of this process (to synchronize). In case of loop, this function also adds the corresponding loop arcs to the Petri net. If a loop is not detected in the control of the parallelism, then the parallelism continues normally as in the standard semantics. If a loop is detected, then a new control labeled with $\circlearrowleft$ is returned directly without performing any subderivation. Another important task performed by this function is the preparation of the store. This function builds the new store indicating what rules must be applied in the following derivations, also for its internal parallelisms.

In order to build the new store, function `AppRules` is used. It returns the set of rules R that can be applied to a synchronized parallelism $P \underset{X}{\|} Q$.

Essentially, `AppRules` decides what rules are applicable depending on the events that could happen in the next step. These events can be inferred by using function `FstEvs`. In particular, given a process P, function `FstEvs` returns the set of events that can trigger a rule in the semantics using P as the control. Observe that `AppRules` implicitly imposes an order in the execution, and this order avoids the repetition of redundant derivations. For instance, if both branches of a parallelism can fire event τ in any order, then it will be fired first in the first branch (using rule `SP1`) and then in the second branch (using rule `SP2`). This avoids multiple unnecessary executions such as `SP1`, `SP2` and `SP2`, `SP1` where only τ happens in both branches but in different order. Therefore, rule (Synchronized Parallelism 5) prepares the store allowing the semantics to proceed with the correct rule.

Example 4. *Consider again the specification of Example 2. Due to the set of synchronized events* $\{\texttt{0}, \texttt{1}, \texttt{divisible3}\}$ *in process* `MAIN` *and to the choice operators in processes* `REM0` *and* `REM1`*, this specification can produce the set of finite sequences of observable events defined as traces*(`MAIN`) *in (2). In particular, it can produce the sequence of events* $\langle \texttt{1}, \texttt{1}, \texttt{0}, \texttt{divisible3} \rangle$ *before it is deadlocked. The execution of Algorithm 1 with Example 2 produces the Petri net shown in Figure 10(a). This Petri net is generated after eight iterations of the algorithm (and thus eight executions of the instrumented semantics). The first two iterations are shown step by step in Figures 7 and 8.*

In these figures, for each state, we show a sequence of rules applied from left to right to obtain the next state. We first execute the semantics with the initial state $(\texttt{MAIN}, p_0, \mathcal{N}_0, ([], []), \emptyset)$ *and get the computation* ***First iteration****. This computation corresponds to the execution of the left branch of the two choices of process* `REM0`*. The final state is* $State_6 = (\texttt{STOP}, p_1, \mathcal{N}_5, ([], S_6), \emptyset)$*. Note that the store* S_6 *contains two pairs* $(\texttt{C1}, \{\texttt{C2}\})$ *to denote that the left branch of the choices has been executed and the right branch is still pending. Then, the algorithm calls function* `UpdStore` *and executes the semantics again with the new initial state* $State_7 = (\texttt{MAIN}, p_0, \mathcal{N}_5, (S_7, []), \emptyset)$ *and it gets the computation* ***Second iteration****. After this execution the Petri net* ($\mathcal{N}_9$) *shown in Figure 9 has been computed. The first iteration generates the white nodes of Figure 9 and grey nodes are generated in the second iteration. Figure 10(a) shows the final Petri net generated where white nodes were generated in the first and second iterations, grey nodes were generated in the third iteration; and black nodes were generated in the rest of iterations (from fourth to eighth).*

The language produced by the labeled Petri net in Figure 10(a) is:

$$L(\mathcal{N}) = \{\langle\rangle, \langle\tau\rangle, \langle\tau,\ ||\rangle, \langle\tau,\ ||,\ \tau\rangle, \langle\tau,\ ||,\ \tau,\ \tau\rangle, \langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C2}\rangle, \langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1}\rangle,$$
$$\langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2}\rangle, \langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C1}\rangle, \langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ 1\rangle,$$
$$\langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ 1,\ \tau\rangle, \langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ 1,\ \tau,\ \texttt{C1}\rangle,$$
$$\langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ 1,\ \tau,\ \texttt{C2}\rangle, \langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ 1,\ \tau,\ \texttt{C2},\ 1\rangle,$$
$$\langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ 1,\ \tau,\ \texttt{C2},\ 1,\ \tau\rangle,$$
$$\langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ 1,\ \tau,\ \texttt{C2},\ 1,\ \tau,\ \texttt{C2}\rangle,$$
$$\langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ 1,\ \tau,\ \texttt{C2},\ 1,\ \tau,\ \texttt{C1}\rangle,$$
$$\langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ 1,\ \tau,\ \texttt{C2},\ 1,\ \tau,\ \texttt{C1},\ \texttt{C2}\rangle,$$
$$\langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ 1,\ \tau,\ \texttt{C2},\ 1,\ \tau,\ \texttt{C1},\ \texttt{C1}\rangle,$$
$$\langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ 1,\ \tau,\ \texttt{C2},\ 1,\ \tau,\ \texttt{C1},\ \texttt{C1},\ 0\rangle,$$
$$\langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ 1,\ \tau,\ \texttt{C2},\ 1,\ \tau,\ \texttt{C1},\ \texttt{C1},\ 0,\ \tau\rangle,$$
$$\langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ 1,\ \tau,\ \texttt{C2},\ 1,\ \tau,\ \texttt{C1},\ \texttt{C1},\ 0,\ \tau,\ \texttt{C1}\rangle,$$
$$\langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ 1,\ \tau,\ \texttt{C2},\ 1,\ \tau,\ \texttt{C1},\ \texttt{C1},\ 0,\ \tau,\ \texttt{C1},\ \texttt{C1}\rangle,$$
$$\langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ 1,\ \tau,\ \texttt{C2},\ 1,\ \tau,\ \texttt{C1},\ \texttt{C1},\ 0,\ \tau,\ \texttt{C1},\ \texttt{C2}\rangle,$$
$$\langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ 1,\ \tau,\ \texttt{C2},\ 1,\ \tau,\ \texttt{C1},\ \texttt{C1},\ 0,\ \tau,\ \texttt{C2}\rangle,$$
$$\langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ 1,\ \tau,\ \texttt{C2},\ 1,\ \tau,\ \texttt{C1},\ \texttt{C1},\ 0,\ \tau,\ \texttt{C2},\ \texttt{divisible3}\rangle\}$$

First iteration

$State_0 = (\texttt{MAIN}, p_0, \mathcal{N}_0, ([], []), \emptyset)$ (PC-Seq)

where $\mathcal{N}_0 = ((\{p_0\}, \emptyset, \emptyset), M_0, \mathcal{P}, \mathcal{T}, \mathcal{L}_P, \mathcal{L}_T), M_0(p_0) = 1,$

$\mathcal{P} = Names \cup \{\square, \sqcap, \texttt{STOP}\}, \mathcal{T} = \Sigma^\tau \cup \{\|, \texttt{C1}, \texttt{C2}\}$

$State_1 = (\texttt{REM0}_\diamond \underset{\{0,1,\texttt{divisible3}\}}{\|}{}_{(\perp,\perp,\emptyset)} \texttt{BINARY}_\diamond, p_1, \mathcal{N}_1, ([], []), \emptyset)$ (SP5)(SP1) (PC-Par)

where $\mathcal{N}_1 = \mathcal{N}_0[p_{\texttt{MAIN}} \mapsto t_\tau \mapsto p_1]$ and $\mathcal{L}_P(p_0) = \texttt{MAIN}$

$State_2 = (rhs(\texttt{REM0}) \underset{\{0,1,\texttt{divisible3}\}}{\|}{}_{lab_2} \texttt{BINARY}_\diamond, p_1, \mathcal{N}_2, ([], S_2), \emptyset)$ (SP5)(SP2) (PC-Par)

where $rhs(\texttt{REM0}) = (((0 \to \texttt{REM0})\square(1 \to \texttt{REM1}))\square(\texttt{divisible3} \to \texttt{STOP})),$

$\mathcal{N}_2 = \mathcal{N}_1[p_1 \mapsto t_\| \mapsto p_2, p_1 \mapsto t_\| \mapsto p_3, p_{\texttt{REM0}} \mapsto t_\tau \mapsto p_4], \mathcal{L}_P(p_2) = \texttt{REM0},$

$lab_2 = (p_4, p_3, \Upsilon_2), \Upsilon_2 = \{(\texttt{REM0}\|\texttt{BINARY}, p_2, p_3)\}$ and $S_2 = [(\texttt{SP1}, \emptyset)]$

$State_3 = (rhs(\texttt{REM0}) \underset{\{0,1,\texttt{divisible3}\}}{\|}{}_{lab_3} rhs(\texttt{BINARY}), p_1, \mathcal{N}_3, ([], S_3), \emptyset)$ (SP5)(SP1) (Choice)

where $rhs(\texttt{BINARY}) = (1 \to 1 \to 0 \to \texttt{divisible3} \to \texttt{STOP}),$

$\mathcal{N}_3 = \mathcal{N}_2[p_{\texttt{BINARY}} \mapsto t_\tau \mapsto p_5], \mathcal{L}_P(p_3) = \texttt{BINARY}, lab_3 = (p_4, p_5, \Upsilon_3),$

$\Upsilon_3 = \Upsilon_2 \cup \{(rhs(\texttt{REM0})\|\texttt{BINARY}, p_4, p_3)\}$ and $S_3 = (\texttt{SP2}, \emptyset) : S_2$

$State_4 = (((0 \to \texttt{REM0})\square(1 \to \texttt{REM1})) \underset{\{0,1,\texttt{divisible3}\}}{\|}{}_{lab_4} rhs(\texttt{BINARY}), p_1, \mathcal{N}_4, ([], S_4), \emptyset)$ (SP5)(SP1) (Choice)

where $\mathcal{N}_4 = \mathcal{N}_3[p_\square \mapsto t_{\texttt{C1}} \mapsto p_6], \mathcal{L}_P(p_4) = \square, lab_4 = (p_6, p_5, \Upsilon_4),$

$\Upsilon_4 = \Upsilon_3 \cup \{(rhs(\texttt{REM0})\|rhs(\texttt{BINARY}), p_4, p_5)\}$ and

$S_4 = [(\texttt{C1}, \{\texttt{C2}\}), (\texttt{SP1}, \emptyset)] : S_3$

$State_5 = ((0 \to \texttt{REM0}) \underset{\{0,1,\texttt{divisible3}\}}{\|}{}_{lab_5} rhs(\texttt{BINARY}), p_1, \mathcal{N}_5, ([], S_5), \emptyset)$ (SP5)(SP4)

where $\mathcal{N}_5 = \mathcal{N}_4[p_\square \mapsto t_{\texttt{C1}} \mapsto p_7], \mathcal{L}_P(p_6) = \square, lab_5 = (p_7, p_5, \Upsilon_5)$

$\Upsilon_5 = \Upsilon_4 \cup \{((((0 \to \texttt{REM0})\square(1 \to \texttt{REM1}))\|rhs(\texttt{BINARY}), p_6, p_5)\}$ and

$S_5 = [(\texttt{C1}, \{\texttt{C2}\}), (\texttt{SP1}, \emptyset)] : S_4$

$State_6 = (\texttt{STOP}, p_1, \mathcal{N}_5, ([], S_6), \emptyset)$ where $S_6 = (\texttt{SP4}, \emptyset) : S_5$

Figure 7: First iteration of Algorithm 1 for Example 2

Second iteration

$State_7 = (\texttt{MAIN}, p_0, \mathcal{N}_5, (\texttt{UpdStore}(S_6), []), \emptyset) = (\texttt{MAIN}, p_0, \mathcal{N}_5, (S_7, []), \emptyset)$ (PC-Seq)

where $S_7 = [(\texttt{C2}, \emptyset), (\texttt{SP1}, \emptyset), (\texttt{C1}, \{\texttt{C2}\}), (\texttt{SP1}, \emptyset), (\texttt{SP2}, \emptyset), (\texttt{SP1}, \emptyset)]$

$State_8 = (\texttt{REM0}_{\diamond} \underset{\{0,1,\texttt{divisible3}\}}{\|}_{(\perp,\perp,\emptyset)} \texttt{BINARY}_{\diamond}, p_1, \mathcal{N}_5, (S_7, []), \emptyset)$ where $\mathcal{L}_P(p_0) = \texttt{MAIN}$ (SP1)(PC-Par)

$State_9 = (rhs(\texttt{REM0}) \underset{\{0,1,\texttt{divisible3}\}}{\|}_{lab_9} \texttt{BINARY}_{\diamond}, p_1, \mathcal{N}_5, (S_8, [(\texttt{SP1}, \emptyset)]), \emptyset)$ (SP2)(PC-Par)

where $rhs(\texttt{REM0}) = (((0 \to \texttt{REM0}) \Box (1 \to \texttt{REM1})) \Box (\texttt{divisible3} \to \texttt{STOP}))$,

$\mathcal{L}_P(p_2) = \texttt{REM0}, lab_9 = (p_4, p_3, \Upsilon_9), \Upsilon_9 = \{(\texttt{REM0} \| \texttt{BINARY}, p_2, p_3)\}$ and

$S_8 = [(\texttt{C2}, \emptyset), (\texttt{SP1}, \emptyset), (\texttt{C1}, \{\texttt{C2}\}), (\texttt{SP1}, \emptyset), (\texttt{SP2}, \emptyset)]$

$State_{10} = (rhs(\texttt{REM0}) \underset{\{0,1,\texttt{divisible3}\}}{\|}_{lab_{10}} rhs(\texttt{BINARY}), p_1, \mathcal{N}_5, (S_9, S_{10}), \emptyset)$ (SP1)(Choice)

where $rhs(\texttt{BINARY}) = (1 \to 1 \to 0 \to \texttt{divisible3} \to \texttt{STOP}), \mathcal{L}_P(p_3) = \texttt{BINARY}$,

$lab_{10} = (p_4, p_5, \Upsilon_{10}), \Upsilon_{10} = \Upsilon_9 \cup \{(rhs(\texttt{REM0}) \| \texttt{BINARY}, p_4, p_3)\}$,

$S_9 = [(\texttt{C2}, \emptyset), (\texttt{SP1}, \emptyset), (\texttt{C1}, \{\texttt{C2}\}), (\texttt{SP1}, \emptyset)]$ and $S_{10} = [(\texttt{SP2}, \emptyset) : (\texttt{SP1}, \emptyset)]$

$State_{11} = (((0 \to \texttt{REM0}) \Box (1 \to \texttt{REM1})) \underset{\{0,1,\texttt{divisible3}\}}{\|}_{lab_{11}} rhs(\texttt{BINARY}), p_1, \mathcal{N}_5, (S_{11}, S_{12}), \emptyset)$ (SP1)(Choice)

where $\mathcal{L}_P(p_4) = \Box, lab_{11} = (p_6, p_5, \Upsilon_{11})$,

$\Upsilon_{11} = \Upsilon_{10} \cup \{(rhs(\texttt{REM0}) \| rhs(\texttt{BINARY}), p_4, p_5)\}$,

$S_{11} = [(\texttt{C2}, \emptyset), (\texttt{SP1}, \emptyset)]$ and $S_{12} = [(\texttt{C1}, \{\texttt{C2}\}), (\texttt{SP1}, \emptyset)] : S_{10}$

$State_{12} = ((1 \to \texttt{REM1}) \underset{\{0,1,\texttt{divisible3}\}}{\|}_{lab_{12}} rhs(\texttt{BINARY}), p_1, \mathcal{N}_6, ([], S_{13}), \emptyset)$ (SP5)(SP3)(Pref)(Pref)

where $\mathcal{N}_6 = \mathcal{N}_5[p_{\Box} \mapsto t_{\texttt{C2}} \mapsto p_8], \mathcal{L}_P(p_6) = \Box, lab_{12} = (p_8, p_5, \Upsilon_{12})$,

$\Upsilon_{12} = \Upsilon_{11} \cup \{(((0 \to \texttt{REM0}) \Box (1 \to \texttt{REM1})) \| rhs(\texttt{BINARY}), p_6, p_5)\}$ and

$S_{13} = [(\texttt{C2}, \emptyset), (\texttt{SP1}, \emptyset)] : S_{12}$

$State_{13} = (\texttt{REM1}_{\diamond} \underset{\{0,1,\texttt{divisible3}\}}{\|}_{lab_{13}} (1 \to 0 \to \texttt{divisible3} \to \texttt{STOP}), p_1, \mathcal{N}_7, ([], S_{14}), \emptyset)$ (SP5)(SP1)(PC-Par)

where $\mathcal{N}_7 = \mathcal{N}_6[p_8 \mapsto t_1 \mapsto p_9, p_5 \mapsto t_1 \mapsto p_{10}], lab_{13} = (p_9, p_{10}, \Upsilon_{13})$,

$\Upsilon_{13} = \Upsilon_{12} \cup \{((1 \to \texttt{REM1}) \| rhs(\texttt{BINARY}), p_8, p_5)\}$

$S_{14} = (\texttt{SP3}, \emptyset) : S_{13}$ and $\Delta_1 = \{(p_8, t_1, p_9), (p_5, t_1, p_{10})\}$

$State_{14} = (rhs(\texttt{REM1}) \underset{\{0,1,\texttt{divisible3}\}}{\|}_{lab_{14}} (1 \to 0 \to \texttt{divisible3} \to \texttt{STOP}), p_1, \mathcal{N}_8, ([], S_{15}), \emptyset)$ (SP5)(SP1)(Choice)

where $rhs(\texttt{REM1}) = ((0 \to \texttt{REM2}) \Box (1 \to \texttt{REM0})), \mathcal{N}_8 = \mathcal{N}_7[p_{\texttt{REM1}} \mapsto t_7 \mapsto p_{11}]$,

$\mathcal{L}_P(p_9) = \texttt{REM1}, lab_{14} = (p_{11}, p_{10}, \Upsilon_{14})$

$\Upsilon_{14} = \Upsilon_{13} \cup \{(\texttt{REM1} \| rhs(\texttt{BINARY}), p_9, p_{10})\}$ and $S_{15} = (\texttt{SP1}, \emptyset) : S_{14}$

$State_{15} = ((0 \to \texttt{REM2}) \underset{\{0,1,\texttt{divisible3}\}}{\|}_{lab_{15}} (1 \to 0 \to \texttt{divisible3} \to \texttt{STOP}), p_1, \mathcal{N}_9, ([], S_{16}), \emptyset)$ (SP5)(SP4)

where $\mathcal{N}_9 = \mathcal{N}_8[p_{\Box} \mapsto t_{\texttt{C1}} \mapsto p_{12}], \mathcal{L}_P(p_{11}) = \Box, lab_{15} = (p_{12}, p_{10}, \Upsilon_{15})$

$\Upsilon_{15} = \Upsilon_{14} \cup \{(rhs(\texttt{REM1}) \| (1 \to 0 \to \texttt{divisible3} \to \texttt{STOP}), p_{11}, p_{10})\}$ and

$S_{16} = [(\texttt{C1}, \{\texttt{C2}\}), (\texttt{SP1}, \emptyset)] : S_{15}$

$State_{16} = (\texttt{STOP}, p_1, \mathcal{N}_9, ([], S_{17}), \emptyset)$ where $S_{17} = (\texttt{SP4}, \emptyset) : S_{16}$

Figure 8: Second iteration of Algorithm 1 for Example 2

If we restrict the language to visible events, we get the language over the alphabet Σ:

$$L_{\Sigma}(\mathcal{N}) = \{\langle\rangle, \langle 1\rangle, \langle 1, 1\rangle, \langle 1, 1, 0\rangle, \langle 1, 1, 0, \texttt{divisible3}\rangle\}$$

We can see that this language is exactly the same as the one produced by

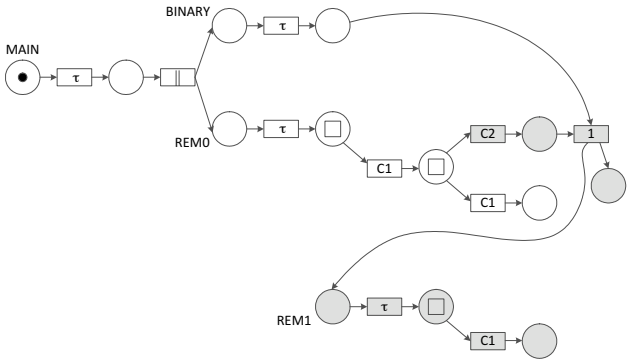


Figure 9: Petri net generated in the **First** and **Second iterations** of Algorithm 1 for Example 2

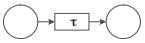
traces(`MAIN`) *in (2). Then, according to Definition 7, the Petri net generated by Algorithm 1 and the CSP specification of Example 2 are equivalent.*

5. Tuning the generated Petri net

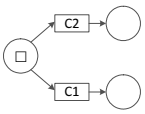
This section introduces a transformation for Petri nets that can be applied to the Petri nets generated by our technique. The transformation takes advantage of the particular shape of the generated Petri nets to remove unnecessary parts that do not contribute to the language produced by them.

Probably, the reader has noticed that the Petri nets generated by Algorithm 1 are very close to the CSP's semantics behavior. These Petri nets follow step by step the sequence of events (both internal and external) that happened during the evaluation of the semantics. For instance,

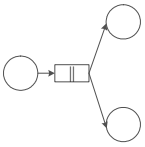
- Each occurrence of a τ is explicitly represented in the Petri net with a transition (labeled with the internal event τ) between two places:

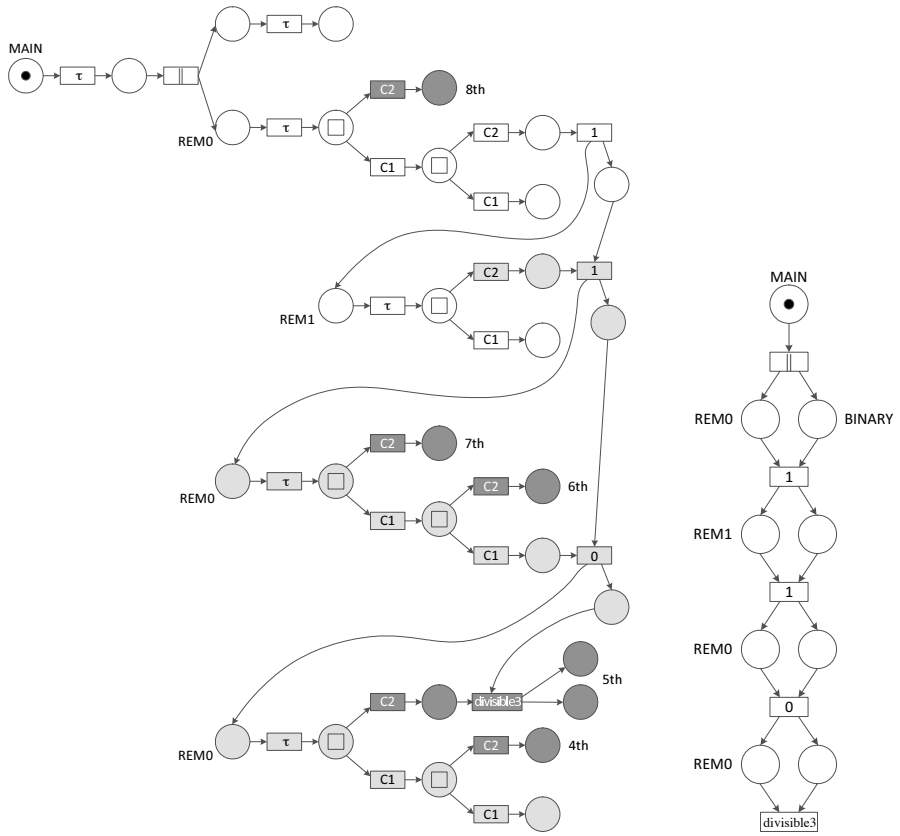


- Each choice is represented with a place labeled with the choice operator ($\Box$ or $\sqcup$) and two transitions labeled with the first option (`C1`) and the second option (`C2`):



- Each parallelism operator is represented with a transition labeled with the parallelism operator || and two places:





(a) PN generated with Algorithm 1 (b) PN optimized

Figure 10: PN associated with the specification of Example 2

These properties makes the generated Petri net to be compositional, and it is easy to see that the Petri net is a graphical representation of the CSP's semantics derivations. In fact, this is a powerful tool for program comprehension because the Petri net is very similar to the so-called *Control Flow Graph* (CFG) (see, e.g., [26]) of imperative languages. Note that the paths followed by tokens are the possible execution paths in the semantics.

However, for some applications, we may be interested in producing a Petri net as small as possible discarding internal events and only concentrating on external ones. This simplified Petri net could keep the equivalence with the CSP specification and significantly reducing its size. However, (part of) the connection with the CSP semantics behavior would be lost.

Because both versions of the Petri net (the complete and the simplified) are useful, we decided not to generate the simplified version directly from the instrumented semantics, and do it with a post-process transformation. This has the additional advantage of not introducing more complexity in the instrumentation of the semantics. As an example consider the complete Petri net in Figure 10(a) and its simplified version in Figure 10(b).

The simplification process of Petri nets is performed by Algorithm 2. This algorithm takes a Petri net and iteratively deletes all parts of the net that are

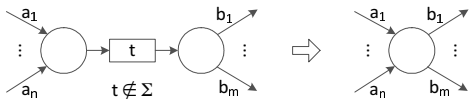
duplicated or useless, until a fix-point is reached (i.e., no more simplifications can be done).

The task of deleting duplicate nodes is performed by function `DelDuplicates` whose implementation can be found in Algorithm 3. The task of deleting useless parts of the Petri net such as sequences of linear non-observable transitions is independent of the previous algorithm. This task is performed by function `DelUseless` and it is implemented in Algorithm 5.

Function `DelDuplicates` traverses the Petri net from the initial place (labeled with `MAIN`) following all paths. During the traversal, it tries to identify repeated parts of the Petri net to remove the repetitions and reuse one of them, whenever it is possible. For this, three auxiliary functions implemented in Algorithm 4 are introduced: `Equal`, `DelNode` and `DelNodes`. Function `Equal` is used to compare two nodes of the Petri net; and functions `DelNode` and `DelNodes` are used to remove the duplicated parts of the Petri net.

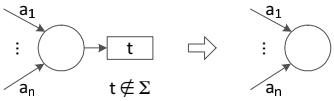
Function `DelUseless` removes useless nodes by checking those transitions that do not contribute to the final trace. These transitions are called *Candidates* and they are initially those transitions labeled with τ, `C1` and `C2`. The function checks whether a sequence of transitions of this kind exists, and if so, they are removed. For instance, some clear opportunities for optimization are the following:

- Removing useless transitions:



- Removing sink places:

- Removing final non-observable transitions:



Example 5. *In Figure 11 we show the optimized Petri net associated with the specification of Example 1. This Petri net is the output produced by Algorithm 2 with the Petri net in Figure 5 as input.*

Algorithm 2 Optimization Algorithm

Input: A labeled Petri net $\mathcal{N} = (\langle P, T, F\rangle, M_0, \mathcal{P}, \mathcal{T}, \mathcal{L}_P, \mathcal{L}_T)$

Output: An optimized labeled Petri net
$\mathcal{N}' = (\langle P', T', F'\rangle, M_0', \mathcal{P}, \mathcal{T}, \mathcal{L}_P', \mathcal{L}_T')$

repeat
$P_{old} = P, T_{old} = T, F_{old} = F$
$(P, T, F, \mathcal{L}_T) =$ `DelDuplicates`$((\langle P, T, F\rangle, M_0, \mathcal{P}, \mathcal{T}, \mathcal{L}_P, \mathcal{L}_T))$
$(P, T, F, \mathcal{L}_T) =$ `DelUseless`$((\langle P, T, F\rangle, M_0, \mathcal{P}, \mathcal{T}, \mathcal{L}_P, \mathcal{L}_T))$
until $P = P_{old} \wedge T = T_{old} \wedge F = F_{old}$

$P' = P, T' = T, F' = F$
$M_0'(p) = M_0(p)\ \forall p \in P'$, $\mathcal{L}_P'(p) = \mathcal{L}_P(p)\ \forall p \in P'$, $\mathcal{L}_T'(t) = \mathcal{L}_T(t)\ \forall t \in T'$

return $\mathcal{N}' = (\langle P', T', F'\rangle, M_0', \mathcal{P}, \mathcal{T}, \mathcal{L}_P', \mathcal{L}_T')$

Algorithm 3 Function `DelDuplicates`

Function `DelDuplicates`$((\langle P, T, F\rangle, M_0, \mathcal{P}, \mathcal{T}, \mathcal{L}_P, \mathcal{L}_T))$
$Visited = \emptyset$, $Pending = \{p_0 \in P \mid \mathcal{L}_P(p_0) =$ `MAIN`$\}$
foreach $n \in Pending$
if $n \in ((P \cup T) \backslash Visited)$
then $Eqs = \{n_{eq} \in P \cup T \mid n_{eq} \neq n \wedge$ `Equal`$(n_{eq}, n, P, T, F, \mathcal{L}_P, \mathcal{L}_T)\}$
if $Eqs = \emptyset$
then $Pending = Pending \cup (n^\bullet \backslash Visited)$
else if $n \in P$
then foreach $n_{eq} \in Eqs$
$F = (F \cup \{(n_p, n) \mid n_p \in {}^\bullet n_{eq}\}) \backslash \{(n_p, n_{eq}) \in F\}$
$(P, T, F) =$ `DelNode`(n_{eq}, P, T, F)
else if $\forall n_{eq} \in Eqs : (\exists n_p \in {}^\bullet n \mid n_{eq} \in n_p^\bullet)$
then $(P, T, F) =$ `DelNodes`$(n_p \in {}^\bullet n, Eqs, P, T, F)$
else if $\nexists n_s \in n^\bullet \mid \forall n_{eq} \in Eqs : n_{eq} \in {}^\bullet n_s$
then $P = P \cup \{p_{new}\}$ where $p_{new} \notin P$
$T = T \cup \{t_{new}\}$ where $t_{new} \notin T$,
$F = F \cup \{(p_{new}, t_{new})\}$
$\mathcal{L}_T(t_{new}) = \mathcal{L}_T(n)$
foreach $t_{eq} \in Eqs \cup \{n\}$
$\mathcal{L}_T(t_{eq}) = \tau$
$(P, T, F) =$ `DelNodes`$(t_{eq}, t_{eq}^\bullet, P, T, F)$
$F = F \cup \{(t_{eq}, p_{new})\}$
$Visited = Visited \cup Eqs$
$Pending = Pending \backslash \{n\}$
$Visited = Visited \cup \{n\}$

return $(P, T, F, \mathcal{L}_T)$

Example 6. *Turning back to the specification in Example 2, its associated Petri net in Figure 10(a) is optimized by Algorithm 2 producing the Petri net in Figure 10(b). Observe that in this simplified Petri net it has been made clearly explicit the parallel execution of process* `BINARY` *with the sequential execution of processes* `REM0` *and* `REM1`*. It is also clear that event* `divisible3` *can only*

Algorithm 4 Functions `DelNode`, `DelNodes` and `Equal`

Function $\texttt{DelNodes}(n_p, Nodes, P, T, F)$
  foreach $n \in Nodes$
    $(P,T,F) = \texttt{DelNode}(n, P, T, F)$
    $F = F \backslash \{(n_p, n)\}$
  return (P,T,F)

Function $\texttt{DelNode}(n, P, T, F)$
  if $n^\bullet = \emptyset \vee n^\bullet = \{n_p\}$
  then $(P,T,F) = \texttt{DelNodes}(n, n^\bullet, P \backslash \{n\}, T \backslash \{n\}, F)$
  return (P,T,F)

Function $\texttt{Equal}(n, n', P, T, F, \mathcal{L}_P, \mathcal{L}_T)$
  return

$$\begin{cases} \text{true} & \text{if } (n \in P \wedge n = n') \vee \\ & \quad ((n^\bullet \cup n'^\bullet) = \emptyset \wedge \texttt{SameLbl}(n, n') \wedge \\ & \quad \texttt{Comparable}(n) \wedge \texttt{Comparable}(n')) \\ \texttt{Eq}(n_s, n'_s) & \text{if } \texttt{SameLbl}(n, n') \wedge \\ & \quad \texttt{Comparable}(n) \wedge \texttt{Comparable}(n') \wedge \\ & \quad n^\bullet = \{n_s\} \wedge n'^\bullet = \{n'_s\} \\ (\texttt{Eq}(n_{s1}, n'_{s1}) \wedge \texttt{Eq}(n_{s2}, n'_{s2})) \vee & \text{if } \texttt{SameLbl}(n, n') \wedge \\ (\texttt{Eq}(n_{s1}, n'_{s2}) \wedge \texttt{Eq}(n_{s2}, n'_{s1})) & \quad \texttt{Comparable}(n) \wedge \texttt{Comparable}(n') \wedge \\ & \quad n^\bullet = \{n_{s1}, n_{s2}\} \wedge n'^\bullet = \{n'_{s1}, n'_{s2}\} \\ \text{false} & \text{otherwise} \end{cases}$$

where $\texttt{Eq}(n, n') = \texttt{Equal}(n, n', P, T, F, \mathcal{L}_P, \mathcal{L}_T)$
  $\texttt{SameLbl}(n, n') = (\nexists \mathcal{L}_P(n) \wedge \nexists \mathcal{L}_P(n')) \vee \mathcal{L}_T(n) = \mathcal{L}_T(n') \vee \mathcal{L}_P(n) = \mathcal{L}_P(n')$
  $\texttt{Comparable}(n) = (n \in P \wedge n^\bullet = \emptyset) \ \vee \ {}^\bullet n = \emptyset \ \vee \ {}^\bullet n = \{n_p\}$

happen if processes `REM0`*,* `REM1`*,* `REM0` *and* `REM0` *are executed sequentially and they synchronize on events* `1`*,* `1`*,* `0` *and* `divisible3` *with the parallel execution of process* `BINARY`*.*

6. Correctness

In this section we state the correctness of the transformation from CSP to Petri nets. In particular, we prove that given a CSP specification, Algorithm 1

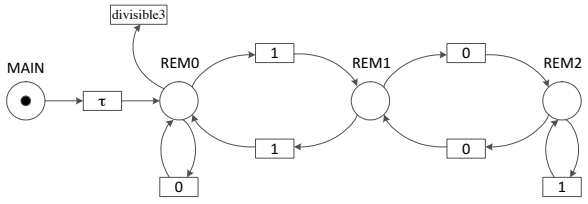


Figure 11: PN optimized associated with the specification of Example 1

Algorithm 5 Function `DelUseless`

Function `DelUseless`$((\langle P, T, F\rangle, M_0, \mathcal{P}, \mathcal{T}, \mathcal{L}_P, \mathcal{L}_T))$
  $Candidates = \{t \in T \mid \mathcal{L}_T(t) \in \{\tau, \texttt{C1}, \texttt{C2}\}\}$
  foreach $t \in Candidates$
    $Candidates = Candidates \backslash \{t\}$
    $NextTrans = \bigcup_{p_{next} \in t^\bullet} p^\bullet_{next}$
    if $(\exists p_{next} \in t^\bullet : \nexists \mathcal{L}_P(p_{next}) \vee \mathcal{L}_P(p_{next}) \in \{\square, \sqcap\}) \wedge$
       $(\nexists p'_{next} \in t^\bullet : \exists t' \in {}^\bullet p'_{next} \wedge t \neq t')$
    then foreach $t_{next} \in NextTrans$
        $F = F \cup \{(p_{prev}, t_{next}) \mid p_{prev} \in {}^\bullet t)\}$
        $F = F \backslash \{(p_{next}, t_{next}) \in F \mid p_{next} \in t^\bullet\}$
      $P = P \backslash t^\bullet$, $T = T \backslash \{t\}$, $F = F \backslash \{(p_{prev}, t), (t, p_{next}) \in F\}$
    else if $(\exists p_{next} \in t^\bullet : \nexists \mathcal{L}_P(p_{next}) \vee \mathcal{L}_P(p_{next}) \in \{\square, \sqcap\}) \wedge$
       $(NextTrans = \{t_{next}\} \wedge \mathcal{L}_T(t_{next}) \notin \{\tau, \texttt{C1}, \texttt{C2}\} \wedge |{}^\bullet t_{next}| = 1)$
      then $PrevTrans = \bigcup_{p_{prev} \in {}^\bullet t_{next}} {}^\bullet p_{prev}$
        foreach $t_{prev} \in PrevTrans$
          $\mathcal{L}_T(t_{prev}) = \mathcal{L}_T(t_{next})$
          $F = F \backslash \{(t_{prev}, p_{prev}) \in F \mid (p_{prev}, t_{next}) \in F\}$
        if $t^\bullet_{next} \neq \emptyset$
        then $F = F \cup \{(t_{prev}, p_{next}) \mid t_{prev} \in PrevTrans \wedge p_{next} \in t^\bullet_{next}\} \backslash t^\bullet_{next}$
        $P = P \backslash {}^\bullet t_{next}$, $T = T \backslash \{t_{next}\}$, $F = F \backslash \{(p, t_{next}) \in F\}$
  $P = P \backslash \{p \in P \mid (\mathcal{L}_P(p) \in \{\square, \sqcap\} \vee \nexists \mathcal{L}_P(p)) \wedge p^\bullet = \emptyset\}$
  $T = T \backslash \{t \in T \mid \mathcal{L}_T(t) \in \{\tau, \texttt{C1}, \texttt{C2}, \|\} \wedge t^\bullet = \emptyset\}$
  $\mathcal{L}_T(t) = \tau,\ \forall t \in T : \mathcal{L}_T(t) = \| \wedge (\nexists p_{s1}, p_{s2} \in t^\bullet \mid p_{s1} \neq p_{s2})$
  return $(P, T, F, \mathcal{L}_T)$

produces in finite time an equivalent Petri net.

We start by proving that Algorithm 1 terminates. For this proof we need some preliminary definitions and lemmas.

Definition 8. *(Rewriting Step, Derivation) Given a state of the semantics s, a* rewriting step *for s, denoted by $s \overset{\Theta}{\rightsquigarrow} s'$, is the transformation of s into s' by using a rule of the CSP semantics. Therefore, $s \overset{\Theta}{\rightsquigarrow} s'$ if and only if a rule of the form $\frac{\Theta}{s \overset{e}{\rightarrow} s'}$ is applicable, where $e \in \Sigma^\tau$ and Θ is a (possibly empty) set of rewriting steps. Given a CSP process s_0, we say that the sequence $s_0 \overset{\Theta_0}{\rightsquigarrow} \ldots \overset{\Theta_n}{\rightsquigarrow} s_{n+1}$, $n \geq 0$, is a* derivation *of s_0 if and only if $\forall\, i, 0 \leq i \leq n, s_i \overset{\Theta_i}{\rightsquigarrow} s_{i+1}$ is a rewriting step. We say that the derivation is* complete *if and only if there is no possible rewriting step for s_{n+1}.*

We will use this definition to prove that no infinite derivation exists. In the following we will refer to the control of state s as $control(s)$.

Lemma 1. *Given a derivation of a state s_0 with the instrumented semantics $s_0 \overset{\Theta_0}{\rightsquigarrow} \ldots \overset{\Theta_n}{\rightsquigarrow} s_{n+1}$, then the number of different parallelism operators appearing in $\bigcup_{0 \leq i \leq n+1} control(s_i)$ is finite.*

Proof. This lemma trivially follows from the fact that we do not allow CSP specifications with recursive parallelism. Therefore, each parallel operator of the specification can only appear once in a derivation. Hence, the lemma follows because the specification is finite (and so the number of different parallel operators). ■

Lemma 2. *Given a CSP specification and a derivation of a state s_0 with the instrumented semantics $s_0 \overset{\Theta_0}{\rightsquigarrow} \dots \overset{\Theta_n}{\rightsquigarrow} s_{n+1}$, where $\nexists s_i, 0 \leq i \leq n+1 : control(s_i) = P \| Q$ then the number of process calls in $control(s_i)$, $0 \leq i \leq n+1$, is finite.*

Proof. We prove this lemma by induction on the length of the derivation showing that the same process call cannot appear more than twice in a derivation and, hence, the number of process calls is finite because CSP specifications are always finite (thus, also the number of different process calls is finite).

(Base case) The number of process calls in $control(s_0)$ and $control(s_1)$ is finite because $control(s_0) = \texttt{MAIN}$ and $control(s_1) = rhs(\texttt{MAIN})$ (rule (Process Call - Sequential) is applied), moreover, trivially, the same process call cannot appear more than twice in $control(s_1)$. The only possibility that process MAIN appears twice is when it is defined as $\texttt{MAIN} = \texttt{MAIN}$. In such a case, the derivation is:

$$\frac{}{(\texttt{MAIN}, p_0, \mathcal{N}_0, \emptyset, (S_0, []), \emptyset) \xrightarrow{\tau} (\texttt{MAIN}, p', \mathcal{N}[p_M \mapsto t_\tau \mapsto p'], (S_0, []), \emptyset)}$$

$$\frac{}{(\texttt{MAIN}, p', \mathcal{N}[p_M \mapsto t_\tau \mapsto p'], (S_0, []), \emptyset) \xrightarrow{\tau} (\circlearrowleft(\texttt{MAIN}), p', \mathcal{N}[p_M \mapsto t_\tau \mapsto p'], (S_0, []), \emptyset)}$$

This derivation is complete and thus the claim follows.

(Induction hypothesis) We assume that the number of process calls in the controls of s_j, $1 \leq j < n+1$ is finite and that no process call has been repeated more than twice.

(Inductive case) We prove now that the same process call does not appear more than twice in the controls of $s_0 \dots s_{j+1}$. We consider the rewriting step $s_j \overset{\Theta_j}{\rightsquigarrow} s_{j+1}$. Because there are no parallelism operators in the derivation (i.e., $\nexists s_i, 0 \leq i \leq n+1 : control(s_i) = P \| Q$), we know that rule (Process Call - Parallel) cannot be applied. If the rule applied in this step is not a (Process Call - Sequential), then the claim follows trivially because no new process call operators can appear in $control(s_{j+1})$. If the rule applied is a (Process Call - Sequential), say $control(s_j) = P$, then we have two possibilities:

- P was not called in the derivation $s_0 \overset{\Theta_0}{\rightsquigarrow} \dots \overset{\Theta_{j-1}}{\rightsquigarrow} s_j$. Then, s_{j+1} contains a new process call that appears for the first time. Thus, the claim follows.

- P was already called in the derivation $s_0 \overset{\Theta_0}{\rightsquigarrow} \dots \overset{\Theta_{j-1}}{\rightsquigarrow} s_j$. Then, $\mathcal{N}$ contains a node labeled with P introduced in the second item of function LoopCheck. In this case we have a rewriting step: $s_k \overset{\Theta_k}{\rightsquigarrow} s_{k+1}$ with $k < j$ and $control(s_k) = P$. The first item of function LoopCheck is executed and thus $control(s_{k+1}) = \circlearrowleft P$. Hence, the computation finishes because no more rules are applicable and the claim follows.

■

In the next lemma we need a notion of size to measure the processes in a CSP specification.

Definition 9. *(Size of CSP expressions) Given a CSP process P, we define the size of P as:*

$$size(P) = \begin{cases} 1 & \text{if } P = N \in Names \text{ or } P = STOP \\ 1 + size(P_1) & \text{if } P = a \rightarrow P_1 \\ 1 + size(P_1) + size(P_2) & \text{if } P = P_1 \sqcap P_2 \text{ or } \\ & \quad P = P_1 \Box P_2 \text{ or } P = P_1 \underset{X}{||} P_2 \end{cases}$$

We will use this notion of size to prove that some rewriting steps always decrease the size of the CSP processes they have in their control.

Lemma 3. *Given a derivation of a state s_0 with the instrumented semantics $\mathcal{D} = s_0 \overset{\Theta_0}{\rightsquigarrow} \dots \overset{\Theta_n}{\rightsquigarrow} s_{n+1}$ where $s_0 = (\texttt{MAIN}, p_0, \mathcal{N}_0, (S_0, []), \emptyset)$, then $\mathcal{D}$ is finite.*

Proof. Firstly, by Lemma 1, we know that the number of parallelism operators in $\mathcal{D}$ is finite. This means that the number of processes executing in parallel is finite. This is an important property, because then, we only have to prove that every parallel process is finite.

Let us consider $\mathcal{D} = s_0 \overset{\Theta_0}{\rightsquigarrow} \dots \overset{\Theta_n}{\rightsquigarrow} s_{n+1}$ as an arbitrary derivation. We need to prove two different properties:

1. n is a finite number.
2. For each rewriting step $s \overset{\Theta}{\rightsquigarrow} s'$ in $\mathcal{D}$, Θ has a finite number of rewriting steps.

We start with the first item. There are two possibilities:

(a) $\nexists s_i, 0 \leq i \leq n+1 : control(s_i) = P \| Q$

(b) $\exists s_i, 0 \leq i \leq n+1 : control(s_i) = P \| Q$

For concreteness, in the following, we represent rewriting steps by only showing the control of each state. This will simplify the explanations while keeping the relevant component needed to prove the finiteness of derivations. Moreover, we also remove subderivations as they are not going to influence the final conclusion (they are considered in item 2 of the proof). Then, according to Lemma 2, in case (a) where no parallelism appears as a control in the derivation, only two scenarios are possible:

- $P_0 \rightsquigarrow^* M \rightsquigarrow^* M \rightsquigarrow \circlearrowleft (M)$, where the last rewriting step corresponds to an application of rule (Process Call - Sequential).
- $P_0 \rightsquigarrow^* \texttt{STOP}$

Both of them have a finite number of rewriting steps, so the claim holds.
In the case that a parallelism appears in some control of the derivation, there are different possibilities:

- $P_0 \rightsquigarrow^* P||Q \rightsquigarrow^* P'||Q' \rightsquigarrow \texttt{STOP}$, where the last rewriting step rule would correspond to the application of rule (Synchronized Parallelism 4). This case occurs, for instance, when $P' = \texttt{STOP}$ and $Q' = \texttt{STOP}$, and also when both branches are waiting to synchronize, but each one with a different event (thus they will never synchronize, i.e., they are in a deadlock, and thus the process is stopped).

- $P_0 \rightsquigarrow^* P||Q \rightsquigarrow^* P||Q \rightsquigarrow \circlearrowleft(P||Q)$, where the last rewriting step rule would correspond to the application of rule (Synchronized Parallelism 5). In this case, the control is repeated, thus we are in a loop. Hence, the parallelism is labeled with $\circlearrowleft$ and it cannot continue.

- $P_0 \rightsquigarrow^* P||Q \rightsquigarrow^* \circlearrowleft(P')||Q' \rightsquigarrow^* \circlearrowleft(P')||\texttt{STOP} \rightsquigarrow \circlearrowleft(P'||\texttt{STOP})$, where the last rewriting step rule would correspond to the application of rule (Synchronized Parallelism 4). Here, one branch is marked as a loop and the other already terminated. Therefore, the whole parallelism is also marked as a loop with $\circlearrowleft$ and it cannot continue.

- $P_0 \rightsquigarrow^* P||Q \rightsquigarrow^* \circlearrowleft(P')||Q' \rightsquigarrow^* \circlearrowleft(P')||\circlearrowleft(Q'') \rightsquigarrow \circlearrowleft(P'||Q'')$, where the last rewriting step rule would correspond to the application of rule (Synchronized Parallelism 4). This case is analogous to the previous one. Here both branches are in a loop and thus the parallelism is marked as a loop and stopped.

- $P_0 \rightsquigarrow^* P||Q \rightsquigarrow^* \circlearrowleft(P')||Q' \rightsquigarrow^* \circlearrowleft(P')||Q'' \rightsquigarrow P'||Q'' \rightsquigarrow^* (\texttt{STOP}$ or $\circlearrowleft(P''||Q'''))$, where the rewriting step $\circlearrowleft(P')||Q'' \rightsquigarrow P'||Q''$ corresponds to the application of rule (Synchronized Parallelism 4). In this case Q'' cannot continue because it is waiting to synchronize with P'. Therefore, the loop label is removed from $\circlearrowleft(P')$, allowing P' and Q'' to continue. As the specification is finite, the number of possible controls is also finite, thus it is impossible to unfold infinite different combinations of controls P' and Q''. This means that (if the computation does not end with $\texttt{STOP}$) eventually some of them will be repeated (i.e. the control will belong to Υ) and it will be labeled with $\circlearrowleft$ thus stopping the computation.

- $P_0 \rightsquigarrow^* P||Q \rightsquigarrow^* \circlearrowleft(P')||Q' \rightsquigarrow \circlearrowleft(P'||Q')$, where the last rewriting step rule would correspond to the application of rule (Synchronized Parallelism 4) and $P'||Q' \in \Upsilon$. Therefore, even though Q' is waiting to synchronize with P', the loop is not unfolded because it was already unfolded, and the same situation has been repeated. Therefore, the whole parallelism is marked as a loop and the computation finishes.

In all possible cases the derivation is complete, thus it has a finite number of rewriting steps. Therefore the claim of the first item holds.

In order to prove the second item, we show that if the rule applied is a (Prefixing), (Process Call - Sequential and Parallel), (Choice) or (Synchronized Parallelism 4), then Θ is empty and the claim follows trivially. If it is a (Synchronized Parallelism 1), (Synchronized Parallelism 2) or (Synchronized Parallelism 5), then Θ only contains one rewriting step, and if it is a (Synchronized Parallelism 3), then Θ only contains two rewriting steps. In the later two cases we have to prove that these rewriting steps do not contain infinite rewriting steps bottom-up. We can

prove this by showing that, eventually, Θ will perform a (Prefixing), (Process Call - Sequential and Parallel), (Choice) or (Synchronized Parallelism 4); and thus, the number of rewriting steps is finite. This can be proved using the size of $control(s)$ and demonstrating that each rewriting step reduces $size(control(s))$. Because the initial size is finite (since the CSP specification is finite), it will eventually reduce to zero.

Let us consider all possible cases. The only possible rules applied are:

(Prefixing), (Process Call - Sequential and Parallel), (Choice), (Synchronized Parallelism 4) If these rules are applied, $\Theta = \emptyset$ and the claim follows trivially.

(Synchronized Parallelism 1 and 2) Let $\Theta = t \overset{\Theta'}{\rightsquigarrow} t'$. Then, $size(control(s)) < size(control(t))$; hence, the claim follows.

(Synchronized Parallelism 3) This case is analogous to the previous one but two rewriting steps are reduced in size, one for each branch of the parallelism.

(Synchronized Parallelism 5) In this case the rule applied is of the form:

$$\frac{(P, p, \mathcal{N}_P, (S'_P, S_0), \_) \xrightarrow{e} (P', p, \mathcal{N}', (S', S'_0), \Delta)}{(P1 \underset{X}{\|}_{(p_1,p_2,\Upsilon)} P2, p, \mathcal{N}, (S, S_0), \_) \xrightarrow{e} (P'', p, \mathcal{N}'', (S'', S''_0), \Delta')} \; e \in \Sigma^\tau$$

Therefore, $size(control(s)) = size(control(t))$, thus the size is not reduced with this rule. If this rule is applied when the store is not empty, there will not exist subderivations and the control will be labeled with $\circlearrowleft$ to denote the loop, thus the rule cannot be applied again. If this rule is applied when the store is empty, then, after its application, the store is never empty. Hence, this rule cannot be applied infinitely, it can only be applied once, and then another rule must be applied before it can be applied again. Therefore, the claim follows because this rule will be applied a finite number of times.

■

Theorem 1. *(Termination) Given a CSP specification $\mathcal{S}$, the execution of Algorithm 1 with $\mathcal{S}$ as input terminates.*

Proof. The semantics is fired by Algorithm 1 a number of times limited by the number of elements in the store. Therefore, if we prove that the store is finite, then the semantics is fired a finite number of times. And, since we know that the semantics always terminates (according to Lemma 3), then, the algorithm also terminates. Hence, in order to prove that Algorithm 1 terminates we have to show that the store never grows infinitely. For this purpose, we have to show first that all executions of the semantics terminate. This is sufficient because function `UpdStore`, which is the only one that also manipulates the store (in addition to the semantics), always either reduces its size or leaves it unchanged. So, as the only rules that change the store are (Synchronized Parallelism) and (Choice), we have to show that there is no derivation which fires these rules infinitely. This follows from Lemma 3, that ensures that no infinite derivation exists. Therefore, no rule is fired infinitely. And, because rules only increase the store with a finite number of elements, then the store never grows infinitely.

Moreover, by Lemma 1 we know that there is a finite number of parallelism operators in each execution of the semantics. Therefore, the number of processes in parallel is finite, thus a finite number of elements is added to the store. ■

Theorem 2. *(Petri net generated) Algorithm 1 generates a labeled Petri net.*

Proof. Algorithm 1 produces the Petri net by using the semantics in Figure 6. Therefore, because the final Petri net is only produced by the semantics, we can prove this theorem by ensuring that:

1. The graph produced by the semantics is a Petri net according to Definition 3.
2. The Petri net generated by the semantics is finite.

We begin with the first item.

1. The first item can be proved by induction on the length of an arbitrary derivation $\mathcal{D} = s_0 \overset{\Theta_0}{\rightsquigarrow} \dots \overset{\Theta_n}{\rightsquigarrow} s_{n+1}$ performed with the semantics proving that the final graph produced is always a Petri net.
   The input of Algorithm 1 is a CSP specification $\mathcal{S}$ with initial process `MAIN` and initial state $s_0 = (\texttt{MAIN}, p_0, \mathcal{N}_0, ([], []), \emptyset)$. The Petri net associated with the initial state (s_0) of the instrumented semantics is $\mathcal{N} = (\langle \{p_0\}, \emptyset, \emptyset \rangle, M_0, \mathcal{P}, \mathcal{T}, \mathcal{L}_P, \mathcal{L}_T)$, $M_0(p_0) = 1$, $\mathcal{P} = Names \cup \{\Box, \sqcap\}$, $\mathcal{T} = \Sigma^\tau \cup \{\|, \texttt{C1}, \texttt{C2}\}$.
   (Base case) The base case is the first step performed by the semantics. The only applicable rule in s_0 is (Process Call - Sequential). This rule uses function LoopCheck and in this step only the second case of LoopCheck is possible. Then, this rule labels p_0 with `MAIN` (i.e., $\mathcal{L}_P(p_0) = \texttt{MAIN}$), adds an internal transition t_τ and a new place p as follows: $\mathcal{N}_0[p_{\texttt{MAIN}} \mapsto t_\tau \mapsto p]$. p is a new place ready to be connected in next steps. Therefore, the resulting graph is a Petri net according to Definition 3.
   (Induction hypothesis) We assume as the induction hypothesis that the graph generated by the semantics after n steps is a Petri net. Let $s_n = (P, p, \mathcal{N}, (S, S_0), \Delta)$.
   (Inductive case) Now we prove that the application of a rule in s_n also produces a Petri net. We analyze all possible cases that correspond to all possible rules of the semantics to be applied in s_n. The applicable rules are:
   - If a (Process Call - Sequential) is applied, then process P is a process call, say M. This rule activates function LoopCheck. This function has two possibilities:
     (a) $\mathcal{N}[q_M \mapsto \_]$. In this case, function LoopCheck only adds an arc between a transition and a place that already exists in the Petri net. Therefore, the resulting graph is a Petri net according to Definition 3.
     (b) Otherwise. In this case, the rule adds an internal transition t_τ and a new place p as follows: $\mathcal{N}[p_{\texttt{p}_{\texttt{prev}}} \mapsto t_\tau \mapsto p]$, i.e. this transition is connected to place p_{prev} that represents the last place created in the applied previous rule. p is a new place ready to be connected in next steps. Therefore, the resulting graph is a Petri net according to Definition 3.

- If a (Process Call - Parallel) is applied then the rule adds an internal transition t_τ and a new place p as follows: $\mathcal{N}[p_{\mathtt{p_{prev}}} \mapsto t_\tau \mapsto p]$, i.e. this transition is connected to place p_{prev} that represents the last place created in the applied previous rule. p is a new place ready to be connected in next steps. Therefore, the resulting graph is a Petri net according to Definition 3.

- If a (Prefixing) is applied (i.e., $P = a \to Q$) then the instrumented semantics adds a new transition t_a and a new place p connected to the previous one p_{prev}: $\mathcal{N}[p_{prev} \mapsto t_a \mapsto p]$. p is a new place ready to be connected in next steps. Therefore, the resulting Petri net is a Petri net according to Definition 3.

- If a (Choice) is applied (i.e., $P = Q \boxminus R$), function SelectBranch is used to produce the new control P' and the new tuple of stores (S', S'_0) by selecting a branch with the information of the store. This function creates a new transition $t \in \{t_{C1}, t_{C2}\}$ depending of the chosen branch (t_{C1} or t_{C2}) that represents the τ event: $\mathcal{N}[p_{\boxminus} \mapsto t_{C1} \mapsto p]$ or $\mathcal{N}[p_{\boxminus} \mapsto t_{C2} \mapsto p]$, where $p_{\square}$ is the new label of the last place created in the applied previous rule. p is a new place ready to be connected in next steps. Therefore, the resulting Petri net is a Petri net according to Definition 3.

- If a (Synchronized Parallelism 1 or 2) is applied (i.e., $P = P_1 \underset{X}{\|} P_2$), both parallel processes can be intertwiningly executed until a synchronized event is found. Therefore, places and transitions for both processes can be added interwoven to the Petri net. The parallelism operator is represented in the Petri net with a transition $t_\|$. This transition is connected to two new places (p_1 and p_2) one for each branch: $\mathcal{N}[p_{prev} \mapsto t_\| \mapsto p_1]$ and $\mathcal{N}[p_{prev} \mapsto t_\| \mapsto p_2]$, where p_{prev} is the last place created in the applied previous rule. Therefore, the resulting Petri net is a Petri net according to Definition 3.

- If a (Synchronized Parallelism 3) is applied (i.e., again $P = P_1 \underset{X}{\|} P_2$), all the events that have been executed in this step must be synchronized, i.e. all the events occurred in the subderivations of P_1 (Δ_1) and P_2 (Δ_2). This is done in the Petri net by removing the transitions that were added in each subderivation ($\{(p \mapsto t \mapsto p') \mid (p, t, p') \in (\Delta_1 \cup \Delta_2)\}$) and connecting all of them with a single transition t_e. Therefore, the resulting Petri net is a Petri net according to Definition 3.

- If a (Synchronized Parallelism 4) is applied (i.e., $P_1 \underset{X}{\|} P_2$), none of the parallel processes can proceed because they already finished, deadlocked or were labeled with a loop $\circlearrowleft$. When one of the branches has been labeled as a loop, the corresponding Petri net only changes if the other branch is not in a loop. In this situation, function DelEdges removes the arcs introduced by rule (Synchronized Parallelism 5) that connect those process calls that were looped, but the set of transitions and the set of places of $\mathcal{N}$ are not changed. Therefore, the resulting Petri net is a Petri net according to Definition 3.

- If a (Synchronized Parallelism 5) is applied (i.e., $P_1 \underset{X}{\|} P_2$), then the store is empty. Similarly to (Process Call - Sequential), this rule only modifies the Petri net to draw the loops when they are detected. The arcs introduced are exactly the same as in (Process Call - Sequential). These arcs join a transition with a place that already exists in the Petri net. Therefore, the resulting Petri net is a Petri net according to Definition 3.

We conclude that the resulting final graph is a Petri net.

2. The second item is trivially proved with Theorem 1. This theorem ensures that the execution of the semantics is finite. Therefore, it will only produce a finite number of places and transitions with each execution of the semantics because each rule of the semantics only adds to the graph a finite number of transitions and places. Moreover, because the semantics is only executed a finite number of times, we can conclude that the final Petri net will be always finite.

■

Now we present the main result that states the correctness of the transformation.

Theorem 3. *(Correctness) Let $\mathcal{S}$ be a CSP specification and $\mathcal{N}$ the Petri net generated by Algorithm 1. Then, $\mathcal{S}$ is equivalent to $\mathcal{N}$.*

Proof. We prove the theorem by distinguishing two possible situations: a derivation with a single sequential process, and a derivation with parallel processes.

In the case of sequential processes, the prove is quite easy because each rule generates the part of the Petri net associated with the CSP rewriting step. The only interesting case is when a loop is found.

We prove this case by induction on the length of an arbitrary derivation $\mathcal{D} = s_0 \overset{\Theta_0}{\rightsquigarrow} \dots \overset{\Theta_n}{\rightsquigarrow} s_{n+1}$ of the semantics. And we show that every rewriting step produces the corresponding part of the Petri net, and a place ready to connect the other parts of the Petri net.

(Base case) The input of Algorithm 1 is a CSP specification $\mathcal{S}$ with initial process `MAIN`. The initial state of the instrumented semantics is a Petri net $\mathcal{N} = (\langle \{p_0\}, \emptyset, \emptyset \rangle, M_0, \mathcal{P}, \mathcal{T}, \mathcal{L}_P, \mathcal{L}_T)$, $M_0(p_0) = 1$, $\mathcal{P} = Names \cup \{\square, \sqcap\}$, $\mathcal{T} = \Sigma^\tau \cup \{\|, \texttt{C1}, \texttt{C2}\}$.

Therefore, the first rule applied is always (Process Call - Sequential). This rule uses function LoopCheck and in this step only the third case of LoopCheck is possible. In this case, the rule adds an internal transition t_τ and a new place p_1 as follows: $\mathcal{N}[p_{\texttt{MAIN}} \mapsto t_\tau \mapsto p_1]$. τ does not belong to the set $traces(\texttt{MAIN})$ either $L_\Sigma(\mathcal{N})$. Therefore, after this step $traces(\texttt{MAIN}) = L_\Sigma(\mathcal{N}) = \emptyset$. Moreover, p_1 is the sink of the token produced by t_τ ready to be connected to new transitions.

(Induction hypothesis) We assume as the induction hypothesis that $traces(\texttt{MAIN}) = L_\Sigma(\mathcal{N})$ after n steps of the semantics ($s_0 \overset{\Theta_0}{\rightsquigarrow} \dots \overset{\Theta_{n-1}}{\rightsquigarrow} s_n$). And also that the $\mathcal{N}$ contains a place where a token will arrive after the sequence of events associated with the n steps of the semantics occurs in $\mathcal{N}$.

(Inductive case) We now prove that after the application of a rule of the semantics in $s_n \overset{\Theta_n}{\rightsquigarrow} s_{n+1}$, $traces(\texttt{MAIN}) = L_\Sigma(\mathcal{N})$ also holds. Let us consider the rewriting step $(P, p, \mathcal{N}, (S, S_0), \Delta) \xrightarrow{\tau} (P', p', \mathcal{N}', (S', S_0'), \Delta')$.

We analyze all possible cases that correspond to all possible rules of the semantics. There are some different rules available:

- If a (Process Call - Sequential) occurs in $\mathcal{S}$, P is a process call M and this rule activates function LoopCheck. There are two possibilities:

  1. If a loop is detected (a place labeled with M already exists in $\mathcal{N}$, i.e., $\mathcal{N}[q_M \mapsto \_]$), then transition t (connected to the current place p) is connected to q_M to form the loop ($\mathcal{N}[t \mapsto q_M]$)). Observe that in this case, no further rewriting steps are possible. Therefore, there is no need for a new place ready for future rewriting steps.
  2. Otherwise, the process call is unfolded normally. In this case, the rule adds an internal transition t_τ and a new place p_1 as follows: $\mathcal{N}[p_{\texttt{MAIN}} \mapsto t_\tau \mapsto p_1]$. τ does not belong to the set $traces(\texttt{MAIN})$ either $L_\Sigma(\mathcal{N})$. Therefore, after this step $traces(\texttt{MAIN}) = L_\Sigma(\mathcal{N})$, i.e., they have not changed. Here again, we have p_1 ready for next steps.

- If a (Prefixing) occurs in $\mathcal{S}$ (i.e., $P = a \to Q$) then the instrumented semantics adds a new transition t_a and a new place p' connected to the previous one p: $\mathcal{N}[p \mapsto t_a \mapsto p']$. Therefore, event a is observable in $traces(\texttt{MAIN})$ and a belongs to $L_\Sigma(\mathcal{N})$. Therefore, after this step it still holds that $traces(\texttt{MAIN}) = L_\Sigma(\mathcal{N})$.

- If a (Choice) occurs in $\mathcal{S}$ (i.e., $P = Q \boxminus R$), function SelectBranch is used to produce the new control P' and the new tuple of stores (S', S'_0) by selecting a branch with the information of the store. This function creates a new transition for the selected branch, either (t_{C1} or t_{C2}) that represents the τ event: $\mathcal{N}[p_\boxminus \mapsto t_{C1} \mapsto p']$ or $\mathcal{N}[p_\boxminus \mapsto t_{C2} \mapsto p']$. $C1$ and $C2$ do not belong to the set $traces(\texttt{MAIN})$. Therefore, after this step it still holds that $traces(\texttt{MAIN}) = L_\Sigma(\mathcal{N})$. And, again, we have a new place p' ready for next rewriting steps.

Therefore, in the case of sequential execution, $traces(\texttt{MAIN}) = L_\Sigma(\mathcal{N})$ after each step. In the case of parallel execution, there are some special cases such as synchronizations that must be considered. We ignore in the following the application of rules (Process Call - Sequential), (Prefixing) and (Choice) because their behavior is completely analogous to the case of sequential execution.

- If a (Process Call - Parallel) occurs in $\mathcal{S}$ then the rule adds an internal transition t_τ and a new place p_1 as follows: $\mathcal{N}[p_{\texttt{MAIN}} \mapsto t_\tau \mapsto p_1]$. τ does not belong to the set $traces(\texttt{MAIN})$ either $L_\Sigma(\mathcal{N})$. Therefore, after this step $traces(\texttt{MAIN}) = L_\Sigma(\mathcal{N})$, i.e., they have not changed.

- If a (Synchronized Parallelism 1 or 2) occurs in $\mathcal{S}$ (i.e., $P_1 \underset{X}{\|} P_2$), both parallel processes can be intertwiningly executed until a synchronized event is found. Therefore, places and transitions for both processes can be added interwoven to the Petri net. The parallelism operator is represented in the Petri net with a transition $t_\|$. This transition is connected to two new places (p'_1 and p'_2) one for each branch: $\mathcal{N}[p \mapsto t_\| \mapsto p'_1]$ and $\mathcal{N}[p \mapsto t_\| \mapsto p'_2]$. $\|$ does not belong to the set $traces(\texttt{MAIN})$ either $L_\Sigma(\mathcal{N})$. Therefore, after this step it still holds that $traces(\texttt{MAIN}) = L_\Sigma(\mathcal{N})$.

- If a (Synchronized Parallelism 3) occurs in $\mathcal{S}$ (i.e., $P_1 \underset{X}{\|} P_2$), all the events that have been executed in this step must be synchronized, i.e. all the events occurred in the subderivations of P_1 (Δ_1) and P_2 (Δ_2). This is done in the Petri net by removing the transitions that were added in each subderivation ($\{(p \mapsto t \mapsto p') \mid (p, t, p') \in (\Delta_1 \cup \Delta_2)\}$) and connecting all of them with a single transition t_e. Therefore, after this step it still holds that $traces(\texttt{MAIN}) = L_\Sigma(\mathcal{N})$.

- If a (Synchronized Parallelism 4) occurs in $\mathcal{S}$ (i.e., $P_1 \underset{X}{\|} P_2$), none of the parallel processes can proceed because they already finished, deadlocked or were labeled with a loop $\circlearrowleft$. When one of the branches has been labeled as a loop, the corresponding Petri net only changes if the other branch is not in a loop. In this situation, function DelEdges removes the edges introduced by rule (Process Call) that connect those process calls that were looped. Therefore, after this step it still holds that $traces(\texttt{MAIN}) = L_\Sigma(\mathcal{N})$.

- If a (Synchronized Parallelism 5) occurs in $\mathcal{S}$ (i.e., $P_1 \underset{X}{\|} P_2$), then the store is empty or the control is in its Υ. This rule can add some loop arcs from a transition to a place which is already in the Petri Net. Therefore, after this step it still holds that $traces(\texttt{MAIN}) = L_\Sigma(\mathcal{N})$.

■

Corollary 1. *Given two CSP specifications $\mathcal{S}_1, \mathcal{S}_2$, and their associated Petri nets $\mathcal{N}_1, \mathcal{N}_2$ generated by Algorithm 1, we have that $traces(\texttt{MAIN}_1) = traces(\texttt{MAIN}_2)$ if and only if $L_\Sigma(\mathcal{N}_1) = L_\Sigma(\mathcal{N}_2)$, where process $\texttt{MAIN}_1$ belongs to $\mathcal{S}_1$ and process $\texttt{MAIN}_2$ belongs to $\mathcal{S}_2$.*

Proof. We base the proof of this corollary on the fact that both $traces(\texttt{MAIN}_1)$ and $traces(\texttt{MAIN}_2)$ are defined as sets. And also both $L_\Sigma(\mathcal{N}_1)$ and $L_\Sigma(\mathcal{N}_2)$ are sets. Therefore, the notion of equivalence in Definition 7 is based on equivalence between sets of sequences.

By Theorem 3 we know that $\mathcal{S}_1$ is equivalent to $\mathcal{N}_1$ thus $traces(\texttt{MAIN}_1) = L_\Sigma(\mathcal{N}_1)$. Similarly, we know that that $\mathcal{S}_2$ is equivalent to $\mathcal{N}_2$ thus $traces(\texttt{MAIN}_2) = L_\Sigma(\mathcal{N}_2)$.

The equality function between sets has the transitive property. Therefore, the claim follows by transitivity because $traces(\texttt{MAIN}_1) = traces(\texttt{MAIN}_2)$ implies that $L_\Sigma(\mathcal{N}_1) = L_\Sigma(\mathcal{N}_2)$ and vice versa. ■

7. Implementation

All the algorithms proposed and the instrumented operational semantics have been implemented and integrated into a tool called *CSP2PN*. This tool allows us to automatically generate a Petri net equivalent to a given CSP specification. The tool has been implemented in Prolog and C. It has about 1800 LOC and generates Petri nets in the standard `PNML` format [21] (it can also generate Petri nets in `dot` and `jpg` formats). Although *CSP2PN* implements the technique described in this paper, the implemented algorithm is much more complex due to efficiency reasons.

In particular, the implementation of the algorithm contains some improvements that significantly speed up the Petri net construction. The most important improvement is to avoid repeated computations. This is done by: (i) state memoization: once a state already explored is reached the algorithm stops this computation and starts with another one; and (ii) skipping already performed computations: computations do not start from `MAIN`, they start from the next non-deterministic state in the execution (this is provided by the information of the store).

The implementation is composed of eight different modules that interact to produce the final Petri net:

Main This is the main module that coordinates all the other modules.

Control Algorithm Implements Algorithm 1 and the data structures needed to communicate with the semantics (e.g., the store).

Semantics Implements the CSP's extended operational Semantics.

Optimization Implements all the optimization technique (Algorithms 2, 3, 4 and 5).

Pretty Printing All the derivations performed by the semantics and the logs of execution are printed with this module.

Graph Generation It produces the final Petri nets with different formats such as DOT and JPG.

PNML Construction This is the only module written in C (the others are written in Prolog). It basically reads the generated Petri net in DOT format and transforms it into a standard PNML Petri net.

Tools It contains common and auxiliary functions and tools used by the other modules.

The implementation, source code and several examples are publicly available at:

`http://users.dsic.upv.es/~jsilva/CSP2PN/`

There is an online version of *CSP2PN* that can be used to test the tool. This online version is publicly available at:

`http://kaz.dsic.upv.es/csp2petri.html`

Figure 12 shows a screenshot of the online version of *CSP2PN*. You can either write down the initial CSP specification or choose one from the list of available examples. Once the Petri net is generated (Generate Petri net) it is possible to visualize it (View Petri net) and to save it as `pnml`, `jpg` or `dot` formats. The same options are available for the optimized Petri net. For instance, the Petri net in Figure 5 has been automatically generated by *CSP2PN* from the CSP specification of Example 1. After the Petri net is generated, we also show the execution log of the instrumented semantics used by the transformation technique. This log allows us to check the different iterations of the algorithm and to follow the execution of the instrumented semantics step by step.

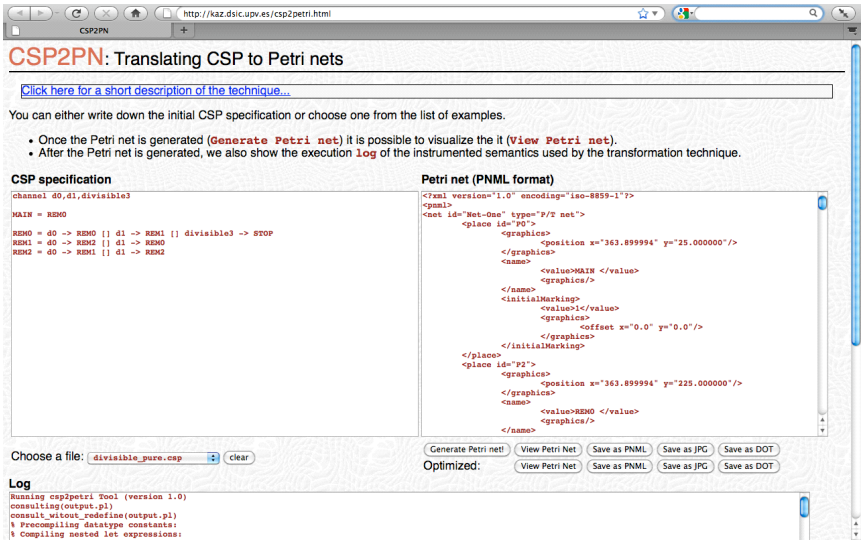


Figure 12: Screenshot of the online version of *CSP2PN*

The possibility of saving the generated Petri net as a `pnml` file allows us to animate and analyze it with any standard Petri net tool. The results of these analyses can be transferred easily to the CSP specification. For instance, *PIPE2* (Platform Independent Petri net Editor 2) [22] is a tool for creating and analysing Petri nets that loads and saves nets in `pnml`. Therefore, the optimized Petri nets generated by *CSP2PN* can be directly verified with the analyses performed by *PIPE2*.

8. Conclusions

This work introduces an algorithm to automatically build a Petri net which produces the same sequences of observable events as a given CSP specification. The algorithm uses an instrumentation of the standard CSP's operational semantics to explore all possible computations of a specification. The semantics is deterministic because the rule applied in every step is predetermined by the initial configuration. Therefore, the algorithm can execute the semantics several times to iteratively explore all computations and hence, generate the whole Petri net. The Petri net is generated even for non-terminating specifications due to the use of a loop detection mechanism controlled by the semantics. This semantics is an interesting result because it explicitly relates the CSP model with the Petri net and the Petri net generated is very similar (structurally) to the CSP specification. The way in which the semantics has been instrumented can be used for other similar purposes with slight modifications. For instance, the same design could be used to generate other graph representations of a computation [12, 13].

The Petri net generated is closely related to the CSP specification because all possible executions force tokens to follow the transitions in such a way that they reproduce the steps of the CSP semantics. This is very interesting compared to previous approaches where the relation between both models is hardly notice-

able. The main cause of this important property is that part of the complexity needed to fill the gap between both models has been translated to the semantics (instead of translating it to the generated Petri net). Hence, an important application of these Petri nets is program comprehension.

However, if we are interested in a reduced version of the Petri net we can further transform it with a transformation defined to remove repeated or unnecessary parts. The resultant Petri nets obtained with this transformation are very compact and can be used to perform different Petri net analyses that can be translated to the CSP specification. Both transformations have been proved correct and terminating.

For future work we plan to extend the set of CSP operators to include sequential composition, parameterized process calls, hiding and renaming. And also, we will study the failures and divergences models in addition to the traces model.

On the practical side, we have implemented a tool called $CSP2PN$ which is able to automatically generate a Petri net equivalent to a CSP specification. The interested reader is referred to: `http://users.dsic.upv.es/~jsilva/CSP2PN` where the implementation, source code and several examples are publicly available.

References

[1] E. Best, R. Devillers and M. Koutny. The Box Algebra = Petri Nets + Process Expressions. Information and Computation 2002; 178:44–100.

[2] J.T. Bradley and W.J. Knottenbelt. The ipc/HYDRA Tool Chain for the Analysis of PEPA Models. Proceedings of the 1st IEEE Conference on the Quantitative Evaluation of Systems (QEST'04), IEEE Computer Society Press, 334–335, 2004.

[3] B. Buth, J. Peleska and H. Shi. Combining Methods for the Livelock Analysis of a Fault-Tolerant System. Proceedings of the 7th International Conference on Algebraic Methodology and Software Technology (AMAST'98), p. 124–139, 1999.

[4] F. de Cindio, G. de Michelis, L. Pomello and C. Simone. A Petri Net Model of CSP. Proceedings of Convención Informática Latina (CIL'81), p. 392–406, Barcelona, 1981.

[5] P. Degano, R. Gorrieri and S. Marchetti. An Exercise in Concurrency: A CSP Process as a Condition/Event System. Advances in Petri Nets 1988, LNCS 340:185–105, Springer-Verlag, 1988.

[6] U. Goltz and W. Reisig. CSP-programs as nets with individual tokens. Advances in Petri Nets 1984, LNCS 188:169–196, Springer-Verlag, 1985.

[7] M. Hack. Petri Net Languages. Technical Report 159, Massachusetts Institute of Technology, Cambridge, MA, USA, 1976.

[8] A. Hall and R. Chapman. Correctness by Construction: Developing a Commercial Secure System. IEEE Software 2002; 19(1):18–25.

[9] J. Hillston. A Compositional Approach to Performance Modelling. PhD thesis, CST-107-94, University of Edinburgh, 1994.

[10] C. A. R. Hoare. Communicating Sequential Processes. Prentice Hall, 1985.

[11] J. E. Hopcroft, R. Motwani, and J. D. Ullman. Introduction to Automata Theory, Languages, and Computation. Pearson/Addison Wesley, 2007.

[12] M. Leuschel, M. Llorens, J. Oliver, J. Silva, and S. Tamarit. The MEB and CEB static analysis for CSP specifications. Post-Proceedings of the 18th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'08), Revised Selected Papers, LNCS 5438:103–118, Springer-Verlag, 2009.

[13] M. Llorens, J. Oliver, J. Silva, and S. Tamarit. An Algorithm to Generate the Context-sensitive Synchronized Control Flow Graph. Proceedings of the 25th ACM Symposium on Applied Computing (SAC 2010), 3:2144–2148, Sierre, Switzerland, 2010.

[14] M. Llorens, J. Oliver, J. Silva, and S. Tamarit. Transforming Communicating Sequential Processes to Petri Nets. In: B.H.V. Topping, J.M. Adam, F.J. Pallarés, R. Bru, M.L. Romero, editors. Proceedings of the Seventh International Conference on Engineering Computational Technology, Civil-Comp Press, Stirlingshire, UK, Paper 26, 2010.

[15] J. Magott. Performance Evaluation of Communicating Sequential Processes (CSP) using Petri Nets. IEE Proceedings-E 1992; 139(3):237-241.

[16] A. Mazzeo, N. Mazzocca, S. Russo, C. Savy and V. Vittorini. Formal Specification of Concurrent Systems: A Structured Approach. The Computer Journal 1998; 41(3):145-162.

[17] R. Milner. A Calculus of Communicating Systems. LNCS 92, Springer-Verlag, 1980.

[18] T. Murata. Petri Nets: Properties, Analysis and Applications. Proceedings of the IEEE 1989, 77(4):541–580, IEEE Computer Society, 1989.

[19] E. R. Olderog. Operational Petri Net Semantics for CCSP. In Advances in Petri Nets 1987, LNCS 266:196–223, Springer-Verlag, 1987.

[20] J. L. Peterson. Petri Net Theory and the Modeling of Systems. Prentice-Hall, 1981.

[21] The Petri Net Markup Language reference site. `http://www.pnml.org/`.

[22] PIPE2: Platform Independent Petri net Editor 2. `http://pipe2.sourceforge.net/`.

[23] M. Ribaudo. Stochastic Petri Net semantics for stochastic process algebras. IEEE Proceedings 6th International Workshop on Petri Nets and Performance Models, 148–157, 1995.

[24] A.W. Roscoe. The Theory and Practice of Concurrency. Prentice-Hall, 2005.

[25] F. T. Sheldon. Specification and Analysis of Stochastic Properties for Concurrent Systems Expressed Using CSP. Ph.D. Dissertation, Computer Science and Engineering Dept, The University of Texas at Arlington, May 1996.

[26] F. Tip. A survey of program slicing techniques. Journal of Programming Languages 1995; 3:121–189.

Dynamic Slicing Techniques for Petri Nets[☆]

Marisa Llorens[a], Javier Oliver[a], Josep Silva[a], Salvador Tamarit[a], German Vidal[a]

[a] *Universitat Politècnica de València, Camino de Vera S/N, E-46022 Valencia, Spain*

Abstract

Petri nets provide a means for modelling and verifying the behavior of concurrent systems. Program slicing is a well-known technique in imperative programming for extracting those statements of a program that may affect a given program point. In the context of Petri nets, computing a net slice can be seen as a graph reachability problem. In this paper, we propose two slicing techniques for Petri nets that can be useful to reduce the size of the considered net, thereby simplifying subsequent analysis and debugging tasks by standard Petri net techniques.

Key words: Petri nets, program slicing, reachability analysis

1. Introduction

Program slicing is a method for decomposing programs in order to extract parts of them—called program *slices*—which are of interest. This technique was first defined by Mark Weiser [20] in the context of program debugging. In particular, Weiser's proposal was aimed at using program slicing for isolating the program staments that may contain a bug, so that finding this bug becomes simpler for the programmer. In general, slicing extracts the statements that may affect some point of interest, referred to as *slicing criterion*.

Let us illustrate this technique with an example taken from [19]. Figure 1(a) shows a simple program which requests a positive integer number n and computes the sum and the product of the first n positive integer numbers. Figure 1(b) shows a slice of this program w.r.t. the slicing criterion (`10`,`product`), i.e., variable `product` in line `10`. As can be seen in the figure, all the computations that do not contribute to the final value of the variable `product` have been removed from the slice.

The work by Weiser has inspired a lot of different approaches to compute slices which include generalizations and concretizations of the initial approach. In general, all of them are classified into two classes: *static* and *dynamic*. A slice is said to be *static* if the input of the program is unknown (this is the case of

[☆]This work has been partially supported by the EU (FEDER) and the Spanish MEC/MICINN under grants TIN2005-09207-C03-02, TIN2008-06622-C03-02, and *Acción Integrada* HA2006-0008.

Email addresses: `mllorens@dsic.upv.es` (Marisa Llorens), `fjoliver@dsic.upv.es` (Javier Oliver), `jsilva@dsic.upv.es` (Josep Silva), `stamarit@dsic.upv.es` (Salvador Tamarit), `gvidal@dsic.upv.es` (German Vidal)

```
(1)  read(n) ;
(2)  i := 1 ;
(3)  sum := 0 ;
(4)  product := 1 ;
(5)  while i <= n do
        begin
(6)        sum := sum + i ;
(7)        product := product * i ;
(8)        i := i + 1 ;
        end ;
(9)  write (sum) ;
(10) write (product) ;
```

(a) Example program.

```
read(n) ;
i := 1 ;

product := 1 ;
while i <= n do
   begin

      product := product * i ;
      i := i + 1 ;
   end ;

write (product) ;
```

(b) Program slice w.r.t. (10,product).

Figure 1: Sub-figures 1(a) and 1(b) show an example of program slicing.

Weiser's approach). On the other hand, it is said to be *dynamic* if a particular input for the program is provided, i.e., a particular computation is considered.

In this work, we propose the use of slicing techniques to produce subnets of a Petri net. A Petri net [13, 14] is a graphic, mathematical tool used to model and verify the behavior of systems that are concurrent, asynchronous, distributed, parallel, non-deterministic and/or stochastic. As a graphic tool, they provide a visual understanding of the system and the mathematical tool facilitates its formal analysis. State space methods are the most popular approach to automatic verification of concurrent systems. In their basic form, these methods explore the transition system associated with the concurrent system. The transition system is a graph, known as the *reachability graph*, that represents the system's reachable states as nodes: there is an arc from one state s to another s', whenever the system can evolve from s to s'. In the worst case, state space methods have to explore all the nodes and transitions in the transition system. This makes the method useless in practice, even though it is simple in concept, due to the state-explosion problem that occurs when a Petri net is applied to non-trivial real problems. The technique is costly even in bounded nets with a finite number of states since, in the worst case, the reachable states are multiplied beyond any primitive recursive function. For this reason, various approaches have been proposed to minimize the number of system states to be studied in a reachability graph [17].

Program slicing has a great potential here since it allows us to syntactically reduce a model in such a way that the reduced model is composed only of those parts that may influence the slicing criterion. Since it was originally defined by Weiser, program slicing has been applied to different formalisms which are not strictly programming languages, like attribute grammars [18], hierarchical state machines [9], Z and CSP-OZ specifications [5, 2, 3], etc. Unfortunately, very little work has been carried out on slicing for Petri nets (some notable exceptions are [4, 11, 15, 16]). For instance, Chang and Wang [4] present a static slicing algorithm for Petri nets that slices out all sets of paths, known as concurrence sets, so that all paths within the same set should be executed concurrently. In [11], a static slicing technique for Petri nets is proposed in

order to divide enormous P/T nets into manageable modules so that the divided model can be analyzed by a compositional reachability analysis technique. A Petri net model is partitioned into concurrent units (Petri net slices) using minimal invariants. In order to preserve all the information in the original model, uncovered places should be added into minimally-connectable concurrent units since minimal invariants may not cover all the places. Finally, in [15, 16], Rakow presents another static slicing technique to reduce the Petri net size and, thus, lessen the problem of state explosion that occurs in the *model checking* [6] of Petri nets [1]. From the best of our knowledge, there is no previous proposal for *dynamic* slicing of Petri nets. This is surprising because considering an initial marking and/or a particular sequence of transition firings would allow us to further reduce the size of the slices and focus on a particular use of the considered Petri net.

In this work, we explore two different alternatives for dynamic slicing of Petri nets. Firstly, we present a slicing technique that extends the slicing criterion in [15, 16] in order to also consider an initial marking. We show that this information can be very useful when analyzing Petri nets and, moreover, it allows us to significantly reduce the size of the computed slice. Furthermore, we show that our algorithm is, in the worst case, as precise as Rakow's algorithm. This can still be seen as a lightweight approach to slicing since its cost is bounded by the number of transitions in the Petri net. Then, we present a second approach that further reduces the size of the computed slice by only considering a particular execution—here, a sequence of transition firings. Clearly, in this case the computed slice is only useful to analyze the considered firing sequence. We illustrate both techniques with examples.

2. Petri Nets

A Petri net [13, 14] is a directed bipartite graph, whose two essential elements are called *places* (represented by circles) and *transitions* (represented by bars or rectangles). The edges of the graph form the *arcs*, which are labelled with a positive integer known as *weight*. Arcs run from places to transitions and vice versa. The *state* of the system modeled by the net is represented by assigning non-negative integers to places. This is known as a *marking*, and is shown graphically by adding small black circles to the places, known as *tokens*. The *dynamic behavior* of the system is simulated by changes in the markings of a Petri net, a process which is carried out by the firing of the transitions. The basic concepts of Petri nets are summarized as follows:

Definition 1. A *Petri net* [13, 14] is a tuple $\mathcal{N} = (P, T, F)$, where:

- P is a set of *places*.
- T is a set of *transitions*, such that $P \cap T = \emptyset$ and $P \cup T \neq \emptyset$.
- F is the *flow relation* that assigns weights to arcs: $F : P \times T \cup T \times P \to \mathbb{N}$.

The *marking* M of a Petri net is defined over the set of places P. For each place $p \in P$ we let $M(p)$ denote the number of tokens contained in p.

A *marked Petri net* Σ is a pair $(\mathcal{N}, M)$ where $\mathcal{N}$ is a Petri net and M is a marking. We denote by M_0 the *initial marking* of the net.

In the following, given a marking M and a set of places P, we denote by $M|_P$ the restriction of M over P, i.e., $M|_P(p) = M(p)$ for all $p \in P$ and $M|_P$ is undefined otherwise.

Definition 2. [14] Given a Petri net $\mathcal{N} = (P, T, F)$, we say that a marking M' *covers* a marking M if $M' \geq M$, i.e., $M'(p) \geq M(p)$ for each $p \in P$.

Given a Petri net $\mathcal{N} = (P, T, F)$, we say that a place $p \in P$ is an *input (resp. output) place* of a transition $t \in T$ iff there is an *input (resp. output) arc* from p to t (resp. from t to p). Given a transition $t \in T$, we denote by ${}^\bullet t$ and $t^\bullet$ the set of all input and output places of t, respectively. Analogously, given a place $p \in P$, we denote ${}^\bullet p$ and $p^\bullet$ the set of all input and output transitions of p, respectively.

Definition 3. Let $\Sigma = (\mathcal{N}, M)$ be a marked Petri net, with $\mathcal{N} = (P, T, F)$. We say that a transition $t \in T$ is *enabled* in M, in symbols $M \xrightarrow{t}$, iff for each input place $p \in P$ of t, we have $M(p) \geq F(p, t)$. A transition may only be fired if it is enabled.

The *firing* of an enabled transition t in a marking M eliminates $F(p, t)$ tokens from each input place $p \in {}^\bullet t$ and adds $F(t, p')$ tokens to each output place $p' \in t^\bullet$, producing a new marking M', in symbols $M \xrightarrow{t} M'$.

We say that a marking M_n is *reachable* from an initial marking M_0 if there exists a *firing sequence* $\sigma = t_1 t_2 \dots t_n$ such that $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \dots \xrightarrow{t_n} M_n$. In this case, we say that M_n is reachable from M_0 through σ, in symbols $M_0 \xrightarrow{\sigma} M_n$. This notion includes the empty sequence ϵ; we have $M \xrightarrow{\epsilon} M$ for any marking M. We say that a firing sequence is *initial* if it starts from an initial marking.

The set of all possible markings which are reachable from an initial marking M_0 in a marked Petri net $\Sigma = (\mathcal{N}, M_0)$ is denoted by $R(\mathcal{N}, M_0)$ (or simply by $R(M_0)$ when $\mathcal{N}$ is clear from the context).

The following notion of *subnet* will be particularly relevant in the context of slicing (roughly speaking, we will identify a slice with a subnet). Let $P' \times T' \cup T' \times P' \subseteq P \times T \cup T \times P$, we say that a flow relation $F' : P' \times T' \cup T' \times P' \to \mathbb{N}$ is a restriction of another flow relation $F : P \times T \cup T \times P \to \mathbb{N}$ over P' and T', in symbols $F|_{(P',T')}$, if F' is defined as follows: $F'(x, y) = F(x, y)$ if $(x, y) \in P' \times T' \cup T' \times P'$ and F' is not defined otherwise.

Definition 4. [8] A *subnet* $\mathcal{N}' = (P', T', F')$ of a Petri net $\mathcal{N} = (P, T, F)$ is a Petri net such that $P' \subseteq P$, $T' \subseteq T$ and F' is a restriction of F over P' and T', i.e., $F' = F|_{(P',T')}$.

3. Dynamic Slicing of Petri Nets

In this section, we introduce our first approach to dynamic slicing of Petri nets. We say that our slicing technique is *dynamic* since an initial marking is taken into account (in contrast to previous approaches, e.g., [4, 11, 15, 16]).

Using an initial marking can be useful, e.g., in debugging. Consider for instance that the user is analyzing a particular trace for a marked Petri net (using a simulation tool [7], which we assume correct), so that an erroneous

state is reached. Here, by *erroneous* state, we mean a marking in which some places have an incorrect number of tokens. In this case, we are interested in extracting the set of places and transitions (more formally, a subnet) that may erroneously contribute tokens to the places of interest, so that the user can more easily locate the bug.

Therefore, our first notion of *slicing criterion* is formalized as follows:

Definition 5. Let $\mathcal{N} = (P, T, F)$ be a Petri net. A *slicing criterion* for $\mathcal{N}$ is a pair $\langle M_0, Q\rangle$ where M_0 is an initial marking for $\mathcal{N}$ and $Q \subseteq P$ is a set of places.

Roughly speaking, given a slicing criterion $\langle M_0, Q\rangle$ for a Petri net $\mathcal{N}$, we are interested in extracting a subnet with those places and transitions of $\mathcal{N}$ which can contribute to change the marking of Q in any execution starting in M_0.

Our notion of *dynamic* slice is defined as follows. In the following, we say that σ' is a *subsequence* of a firing sequence σ w.r.t. a set of transitions T if σ' contains all transitions of σ that belong to T and in the same order.

Definition 6. Let $\mathcal{N} = (P, T, F)$ be a Petri net and let $\langle M_0, Q\rangle$ be a slicing criterion for $\mathcal{N}$. Given a Petri net $\mathcal{N}' = (P', T', F')$, we say that $\mathcal{N}'$ is a slice of $\mathcal{N}$ w.r.t. $\langle M_0, Q\rangle$ if the following conditions hold:

- the Petri net $\mathcal{N}'$ is a subnet of $\mathcal{N}$ and
- for each firing sequence $\sigma = t_1 \dots t_n$, for $\mathcal{N}$, with $M_0 \xrightarrow{t_1} \dots \xrightarrow{t_{n-1}} M_{n-1} \xrightarrow{t_n} M_n$ such that $M_{n-1}(p) < M_n(p)$ for some $p \in Q$, there exists a firing sequence σ' for $(\mathcal{N}', M'_0)$, with $M'_0 = M_0|_{P'}$, such that
  - σ' is a subsequence of σ w.r.t. T',
  - $M'_0 \xrightarrow{\sigma'} M'_m$, $m \leq n$, and
  - M'_m covers $M_n|_{P'}$ (i.e., $M'_m \geq M_n|_{P'}$).

Intuitively speaking, a Petri net $\mathcal{N}'$ is a slice of another Petri net $\mathcal{N}$ if $\mathcal{N}'$ is a subnet of $\mathcal{N}$ (i.e., no additional places nor transitions are added) and the behaviour of $\mathcal{N}$ is preserved in $\mathcal{N}'$ for the restricted sets of places and transitions. In order to formalize this second condition, we require that, for all firing sequences $\sigma = t_1 \dots t_n$ that may *move* tokens to the places of the slicing criterion, i.e.,

$$M_0 \xrightarrow{t_1} \dots \xrightarrow{t_{n-1}} M_{n-1} \xrightarrow{t_n} M_n \;\text{ and }\; M_{n-1}(p) < M_n(p),\ p \in Q$$

the *restriction* of this firing sequence can also be performed on the slice $\mathcal{N}'$, i.e.,

$$M'_0 \xrightarrow{\sigma'} M'_m \;\text{ and }\; M'_m \geq M_n$$

Trivially, given a Petri net $\mathcal{N}$, the complete net $\mathcal{N}$ is always a correct slice w.r.t. any slicing criterion. The challenge then is to produce a slice as small as possible.

Algorithm 1 describes our method to extract a dynamic slice from a Petri net. Intuitively speaking, Algorithm 1 constructs the slice of a Petri net (P, T, F) for a set of places $Q \subseteq P$ as follows. The key idea is to capture a possible token flow relevant for places in Q. For this purpose,

Algorithm 1 Dynamic slicing of a marked Petri net.

Let $\mathcal{N} = (P, T, F)$ be a Petri net and let $\langle M_0, Q\rangle$ be a slicing criterion for $\mathcal{N}$. First, we compute a *backward slice* similar to that of [15]. This is obtained from $\mathsf{b\_slice}_{\mathcal{N}}(Q, \{\,\})$, where function $\mathsf{b\_slice}_{\mathcal{N}}$ is defined as follows:

$$\mathsf{b\_slice}_{\mathcal{N}}(W, W_{done}) = \begin{cases} \{\,\} & \text{if } W = \{\,\} \\ T \cup {}^\bullet T \cup \mathsf{b\_slice}_{\mathcal{N}}(W \setminus W'_{done}, W'_{done}) & \\ \qquad \text{if } W \neq \{\,\}, \text{ where } T = {}^\bullet p, \text{ and } W'_{done} = W_{done} \cup \{p\} & \\ \qquad\qquad \text{for some } p \in P & \end{cases}$$

Now, we compute a *forward slice* from

$$\mathsf{f\_slice}_{\mathcal{N}}(\{p \in P \mid M_0(p) > 0\}, \{\,\}, \{t \in T \mid M_0 \xrightarrow{t}\})$$

where function $\mathsf{f\_slice}_{\mathcal{N}}$ is defined as follows:

$$\mathsf{f\_slice}_{\mathcal{N}}(W, R, V) = \begin{cases} W \cup R & \text{if } V = \{\,\} \\ \mathsf{f\_slice}_{\mathcal{N}}(W \cup V^\bullet, R \cup V, V') & \\ \quad \text{if } V \neq \{\,\}, \text{ where } V' = \{t \in T \setminus (R \cup V) \mid {}^\bullet t \subseteq W \cup V^\bullet\} & \end{cases}$$

Then, the dynamic slice is finally obtained from the intersection of the backward and forward slices. Formally, let

$$P' \cup T' = \mathsf{b\_slice}_{\mathcal{N}}(Q, \{\,\}) \cap \mathsf{f\_slice}_{\mathcal{N}}(\{p \in P \mid M_0(p) > 0\}, \{\,\}, \{t \in T \mid M_0 \xrightarrow{t}\})$$

with $P' \subseteq P$ and $T' \subseteq T$, the computed slice is

$$\mathcal{N}' = (P', T', F|_{(P', T')})$$

- we first compute the possible paths which lead to the slicing criterion,
- then we also compute the paths that may be followed by the tokens of the initial marking.

This can be done by taking into account that (i) the marking of a place p depends on its input and output transitions, (ii) a transition may only be fired if it is enabled, and (iii) the enabling of a transition depends on the marking of its input places. The algorithm is divided in three steps:

- The first step is a backward slicing method (which is similar to the *basic slicing algorithm* of [15]) that obtains a slice $\mathcal{N}_1 = (P_1, T_1, F_1)$ defined as the subnet of $\mathcal{N}$ that includes all input places of all transitions connected to any place p in P_1, starting with $Q \subseteq P_1$.
  - The core of this method is the auxiliary function $\mathsf{b\_slice}_{\mathcal{N}}$, which is initially called with the set of places Q of the slicing criterion together with an empty set of places.
  - For a particular non-empty set of places W and a particular place $p \in W$, function $\mathsf{b\_slice}_{\mathcal{N}}$ returns the transitions T in ${}^\bullet p$ and the input places of these transitions ${}^\bullet T$. Then, function $\mathsf{b\_slice}_{\mathcal{N}}$ moves backwards adding the place p to the set W_{done} and removing from W the updated set W_{done} until the set W becomes empty.

- The second step is a forward slicing method that obtains a slice $\mathcal{N}_2 = (P_2, T_2, F_2)$ defined as the subnet of $\mathcal{N}$ that includes all transitions initially enabled in M_0 as well as those transitions connected as output transitions of places in P_2, starting with $p \in P$ such that $M_0(p) > 0$.
  - We define an auxiliary function $\mathsf{f\_slice}_{\mathcal{N}}$, which is initially called with the places that are marked at M_0, an empty set of transitions and the enabled transitions in M_0.
  - For a particular set of places W, a particular set of transitions R and a particular non-empty set of transitions V, function $\mathsf{f\_slice}_{\mathcal{N}}$ moves forwards adding the places in $V^\bullet$ to W, adding the transitions in V to R and replacing the set of transitions V by a new set V' in which are included the transitions that are not in $R \cup V$ and whose input places are in $W \cup V^\bullet$.
  - Finally, when V is empty, function $\mathsf{f\_slice}_{\mathcal{N}}$ returns the accumulated set of places and transitions $W \cup R$.
- Finally, the third step obtains the slice $\mathcal{N}' = (P', T', F')$ defined as the subnet of $\mathcal{N}$ where P' is the intersection of P_1 and P_2, T' is the intersection of T_1 and T_2, and F' is the restriction of F over P' and T', i.e., the intersection of backward and forward slices.

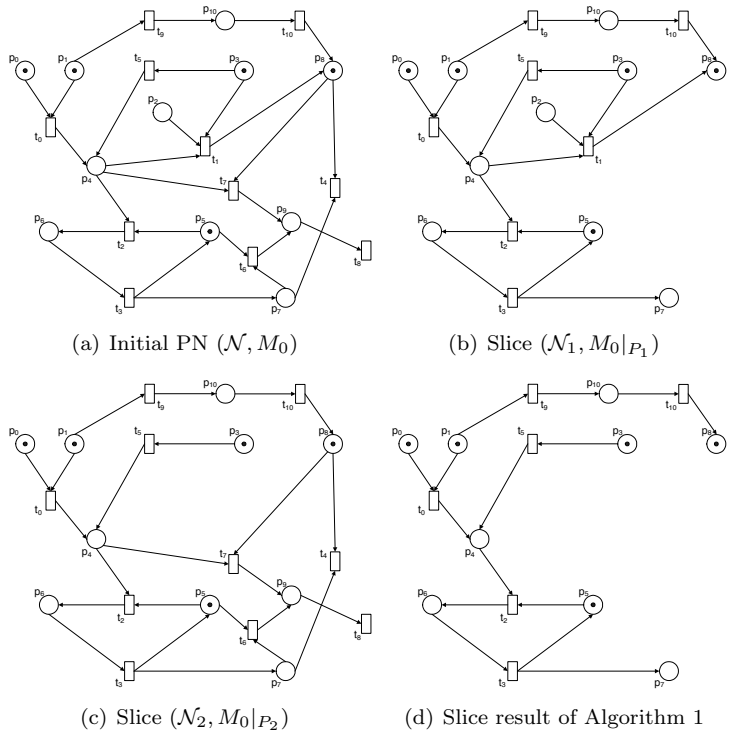
The following result states the completeness of our algorithm for computing Petri net slices. The proof of this result follows easily by induction on the length of the firing sequences considered in Definition 6.

Theorem 1. *Let $\mathcal{N}$ be a Petri net and $\langle M_0, Q\rangle$ be a slicing criterion for $\mathcal{N}$. The dynamic slice $\mathcal{N}'$ computed in Algorithm 1 is a correct slice according to Definition 6.*

We will now show the usefulness of the technique with a simple example.

Example 2. *Consider the Petri net $\mathcal{N}$ of Fig. 2(a) where the user wants to produce a slice w.r.t. the slicing criterion $\langle M_0, \{p_5, p_7, p_8\}\rangle$. Figure 2(b) shows the slice $\mathcal{N}_1$ obtained in the first part of Algorithm 1. Figure 2(c) shows the slice $\mathcal{N}_2$ obtained in the second part of Algorithm 1. The subnet shown in Fig. 2(d) is the final result of Algorithm 1 (the intersection of $\mathcal{N}_1$ and $\mathcal{N}_2$). This slice contains all the places and transitions of the original Petri net which can transmit tokens to the slicing criterion.*

Clearly, using an initial marking allows us to produce smaller slices. Surprisingly, previous approaches completely ignored the marking of the net, and thus their slices are often rather big. For instance, the slice of Fig. 2(b) is a subset of the slice produced by Rakow's algorithm [15] (this algorithm would also include transitions t_4, t_6 and t_7). Clearly, this slice contains parts of the Petri net that cannot be reached with the given initial marking (e.g., transition t_1 which could never be fired because place p_2 is empty). Rakow's algorithm computes all the parts of the Petri net which could transmit tokens to the slicing criterion and, thus, the associated slicing criterion is just $\langle Q\rangle$, where $Q \subseteq P$ is a set of places. In contrast, we compute all the parts of the Petri net which could transmit tokens to the slicing criterion from the initial marking. Therefore, our



(a) Initial PN $(\mathcal{N}, M_0)$ (b) Slice $(\mathcal{N}_1, M_0|_{P_1})$

(c) Slice $(\mathcal{N}_2, M_0|_{P_2})$ (d) Slice result of Algorithm 1

Figure 2: Example of an application of Algorithm 1

technique is essentially a generalization of Rakow's technique because the slice produced with Rakow's algorithm w.r.t. $\langle Q \rangle$ is the same as the slice produced w.r.t. $\langle M_0, Q \rangle$ if $M_0(p) > 0$ for all $p \in P$ and all $t \in T$ are enabled transitions at M_0.

Our slicing technique is more general than Rakow's technique but, at the same time, it keeps its simplicity and efficiency because we still use the Petri net structure to produce the slice. Therefore, our first approach can be considered *lightweight* because its cost is bounded by the number of transitions T of the original Petri net; namely, the cost of our algorithm is $\mathcal{O}(2T)$.

4. Extracting Slices from Traces

In this section, we present an alternative approach to dynamic slicing that generally produces smaller slices by also considering a particular firing sequence.

In principle, Algorithm 1 should consider all possible executions of the Petri net starting from the initial marking. This approach can be useful in some contexts but it is too imprecise for debugging when a particular simulation has been performed. Therefore, in our second approach, we refine the notion of slicing criterion so as to also include the firing sequence that represents the erroneous simulation. By exploting this additional information, the new slicing algorithm will usually produce smaller slices. Formally,

Definition 7. Let $\mathcal{N} = (P, T, F)$ be a Petri net. A *slicing criterion* for $\mathcal{N}$ is

a triple $\langle M_0, \sigma, Q\rangle$ where M_0 is a marking for $\mathcal{N}$, σ is an initial firing sequence (i.e., starting from M_0) and $Q \subseteq P$ is a set of places.

Roughly speaking, given a slicing criterion $\langle M_0, \sigma, Q\rangle$ for a Petri net, we are interested in extracting a subnet with those places and transitions which are necessary to move tokens to the places in Q.

Our notion of *dynamic* slice is defined as follows:

Definition 8. Let $\mathcal{N} = (P, T, F)$ be a Petri net. Let $\langle M_0, \sigma, Q\rangle$ be a slicing criterion for $\mathcal{N}$, with $\sigma = t_1 t_2 \dots t_n$. Given a Petri net $\mathcal{N}' = (P', T', F')$, we say that $\mathcal{N}'$ is a slice of $\mathcal{N}$ w.r.t. $\langle M_0, \sigma, Q\rangle$ if the following conditions hold:

- the Petri net $\mathcal{N}'$ is a subnet of $\mathcal{N}$,
- the set of places Q appears in P' (i.e., $Q \subseteq P'$), and
- there exists a firing sequence σ' for $(\mathcal{N}', M_0')$, with $M_0' = M_0|_{P'}$, such that
  - σ' is a subsequence of σ w.r.t. T',
  - $M_0' \xrightarrow{\sigma'} M_m'$, $m \leq n$, and
  - M_m' covers $M_n|_{P'}$ (i.e., $M_m' \geq M_n|_{P'}$).

Trivially, given a marked Petri net $(\mathcal{N}, M_0)$, the complete net $\mathcal{N}$ is always a correct slice w.r.t. any slicing criterion. The challenge then is to produce a slice as small as possible.

Algorithm 2 Extracting slices from traces.

Let $\mathcal{N} = (P, T, F)$ be a Petri net and let $\langle M_0, \sigma, Q\rangle$ be a slicing criterion for $\mathcal{N}$, with $\sigma = t_1 t_2 \dots t_n$. Then, we compute a dynamic slice $\mathcal{N}'$ of $\mathcal{N}$ w.r.t. $\langle M_0, \sigma, Q\rangle$ as follows:

- We have $\mathcal{N}' = (P', T', F')$, where $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \dots \xrightarrow{t_n} M_n$, $P' \cup T' = \mathsf{slice}(M_n, \sigma, Q)$, $P' \subseteq P$, $T' \subseteq T$, and $F' = F|_{(P',T')}$. Auxiliary function slice is defined as follows:

$$\mathsf{slice}(M_i, \sigma, W) = \begin{cases} W & \text{if } i = 0 \\ \mathsf{slice}(M_{i-1}, \sigma, W) & \text{if } \forall p \in W.\ M_{i-1}(p) \geq M_i(p),\ i > 0 \\ \{t_i\} \cup \mathsf{slice}(M_{i-1}, \sigma, W \cup {}^\bullet t_i) & \text{if } \exists p \in W.\ M_{i-1}(p) < M_i(p),\ i > 0 \end{cases}$$

- The initial marking M_0' is the restriction of M_0 over P', i.e., $M_0' = M_0|_{P'}$.

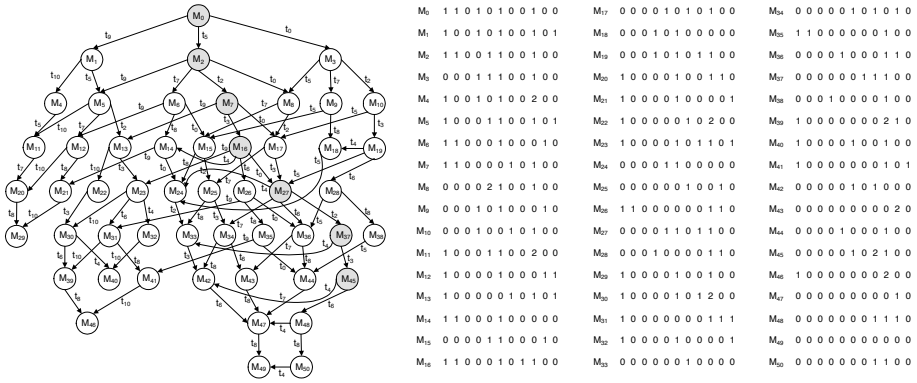
Intuitively speaking, given a slicing criterion $\langle M_0, \sigma, Q\rangle$, the slicing algorithm proceeds as follows:

- The core of the algorithm lies in the auxiliary function slice, which is initially called with the marking M_n which is reachable from M_0 through σ, together with the firing sequence σ and the set of places Q of the slicing criterion.
- For a particular marking M_i, $i > 0$, a firing sequence σ and a set of places W, function slice just moves "backwards" when no place in W increased its tokens by the considered firing.

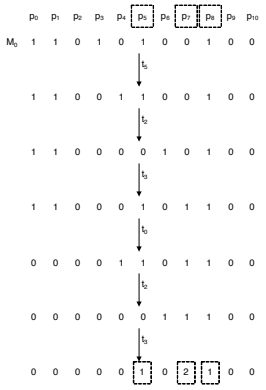
- Otherwise, the fired transition t_i increased the number of tokens of some place in W. In this case, function slice already returns this transition t_i and, moreover, it moves backwards also adding the places in ${}^\bullet t_i$ to the previous set W.

- Finally, when the initial marking is reached, function slice returns the accumulated set of places (which includes the initial places in Q).

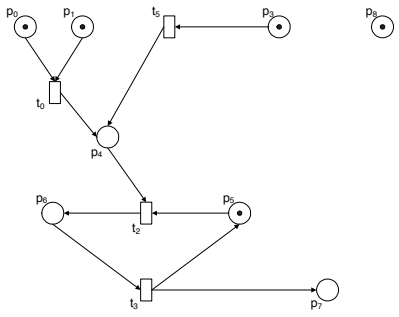
We will now show the utility of the technique with a simple example.



(a) Reachability graph



(b) Firing sequence σ



(c) Slice of $\mathcal{N}$ w.r.t. $\langle M_0, \sigma, \{p_5, p_7, p_8\}\rangle$

Figure 3: Example of an application of Algorithm 2

Example 3. *Consider the Petri net $\mathcal{N}$ of Example 2 shown in Fig. 2(a), together with the firing sequence σ shown in Fig. 3(b). The firing sequence $\sigma = t_5t_2t_3t_0t_2t_3$ corresponds to the branch of the reachability graph shown in Fig. 3(a) that goes from the root to the node M_{45}. Then, the user can define the slicing criterion $\langle M_0, \sigma, \{p_5, p_7, p_8\}\rangle$ for $\mathcal{N}$; where M_0 is the initial marking for $\mathcal{N}$ defined in Fig 2(a).*

Clearly, this slicing criterion focus on a particular execution and thus the slice produced is more precise than the one produced by Algorithm 1. In this case, the slice of $\mathcal{N}$ w.r.t. $\langle M_0, \sigma, \{p_5, p_7, p_8\}\rangle$ is the Petri net shown in Fig. 3(c).

The following result states the completeness of our algorithm for computing Petri net slices.

Theorem 4. *Let $\mathcal{N} = (P, T, F)$ be a Petri net and let $\langle M_0, \sigma, Q\rangle$ be a slicing criterion for $\mathcal{N}$. The dynamic slice $\mathcal{N}'$ computed in Algorithm 2 is a correct slice according to Definition 8.*

Proof. (Sketch) We prove the claim by induction on the number n of transitions in σ.

If $n = 0$, then $\mathsf{slice}(M_0, \sigma, Q) = \bigcup_{p\in Q} \mathsf{slice}(M_0, \sigma, \{p\}) = Q$ and the claim follows trivially for $\mathcal{N}' = (Q, \{\}, \{\})$ and $M_0' = M_0|_Q$.

If $n > 0$, then we distinguish two cases:

- If $M_{n-1}(p) \geq M_n(p)$ for all $p \in Q$, then $\mathsf{slice}(M_n, \sigma, Q) = \mathsf{slice}(M_{n-1}, \sigma, Q)$ and the claim follows by induction.
- Otherwise, there exists some $p \in Q$ with $M_{n-1}(p) < M_n(p)$ and, therefore, $\mathsf{slice}(M_n, \sigma, Q) = \{t_n\} \cup \mathsf{slice}(M_{n-1}, \sigma, Q \cup {}^\bullet t_n)$. Let $\mathcal{N}' = (P', T', F')$, $F' = F|_{(P',T')}$, and $M_0' = M_0|_{P'}$. Now, we prove that $\mathcal{N}'$ is a slice of $\mathcal{N}$ w.r.t. $\langle M_0, \sigma, Q\rangle$:
  - Trivially, $\mathcal{N}'$ is a subnet of $\mathcal{N}$, M_0' is a restriction of M_0 and $Q \subseteq P'$.
  - Let $\mathcal{N}''$ be the slice of $\mathcal{N}$ w.r.t. $\langle M_0, \sigma_{n-1}, Q \cup {}^\bullet t_n\rangle$, with $\sigma_{n-1} = M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \ldots M_{n-1}$ and $\mathcal{N}'' = (P'', T'', F'')$.

    By the inductive hypothesis, there exists a firing sequence σ'' for $(\mathcal{N}'', M_0'')$, with $M_0'' = M_0|_{P''}$, such that
    - σ'' is a subsequence of σ_{n-1} w.r.t. T'',
    - $M_0'' \xrightarrow{\sigma''} M_k''$, $k \leq n-1$, and
    - M_k'' covers M_{n-1} (i.e., $M_k'' \geq M_{n-1}$).

    Now, we consider a firing sequence σ' for $(\mathcal{N}', M_0')$ that mimicks σ'' (which is safe since $P'' = P'$ and $T'' \subseteq T'$) and then adds one more firing depending on whether $t_n \in T'$ or not. If $\sigma' = \sigma''$ then the claim follows by induction. Otherwise, it follows trivially by the inductive hypothesis and the fact that M_k'' covers M_n.

5. Conclusions and Future Work

In this work, we have introduced two different techniques for dynamic slicing of Petri nets. To the best of our knowledge, this is the first approach to dynamic slicing for Petri nets. The first approach takes into account the Petri net and an initial marking, but produces a slice w.r.t. any possibly firing sequence. The second approach further reduces the computed slice by fixing a particular firing sequence. In general, our slices are smaller than previous (static) approaches where no initial marking nor firing sequence were considered.

As a future work, we plan to carry on an experimental evaluation of our slicing techniques in order to test its viability in practice. We also find it useful to extend our slicing technique to other kind of Petri nets (e.g., coloured Petri nets [10] and marked-controlled reconfigurable nets [12]).

References

[1] A. Bell and B.R. Haverkort. Sequential and distributed model checking of Petri nets. *Int. Journal on Software Tools for Technology Transfer*, 7(1):43–60, 2005.

[2] I. Brückner. Slicing CSP-OZ Specifications. In P. Pettersson and W. Yi, editors, *Proc. of the 16th Nordic Workshop on Programming Theory*, number 2004-041 in Technical Reports of the Department of Information Technology, pages 71–73. Uppsala University, Sweden, 2004.

[3] I. Brückner and H. Wehrheim. Slicing Object-Z Specifications for Verification. In *Proc. of the 4th Int'l Conf. of B and Z Users (ZB 2005)*, pages 414–433. Springer LNCS 3455, 2005.

[4] C.K. Chang and H. Wang. A Slicing Algorithm of Concurrency Modeling Based on Petri Nets. In *Proc. of the Int'l Conf. on Parallel Processing (ICPP'86)*, pages 789–792. IEEE Computer Society Press, 1986.

[5] J. Chang and D. Richardson. Static and dynamic specification slicing. In *Proc. of the Fourth Irvine Software Symposium*. Irvine, CA, 1994.

[6] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press: Cambridge, MA, 2000.

[7] Petri Nets Tool Database. Available at `http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db.html`.

[8] J. Desel and J. Esparza. *Free choice Petri nets*. Cambridge University Press, New York, NY, USA, 1995.

[9] M.P.E. Heimdahl and M.W. Whalen. Reduction and Slicing of Hierarchical State Machines. In M. Jazayeri and H. Schauer, editors, *Proc. of the 6th European Software Engineering Conference (ESEC/FSE'97)*, pages 450–467. Springer LNCS 1301, 1997.

[10] K. Jensen. Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1: Basic Concepts, 1992. Volume 2: Analysis Methods, 1994. Volume 3: Practical Use, 1997. Monographs in Theoretical Computer Science, Springer-Verlag.

[11] W.J. Lee, S.D. Cha, Y.R. Kwon, and H.N. Kim. A Slicing-based Approach to Enhance Petri Net Reachability Analysis. *Journal of Research and Practice in Information Technology*, 32(2):131–143, 2000.

[12] M. Llorens and J. Oliver. Introducing Structural Dynamic Changes in Petri Nets: Marked-Controlled Reconfigurable Nets. In Farn Wang, editor, *Proc. of the 2nd Int'l Conf. on Automated Technology for Verification and Analysis (ATVA'04)*, pages 310–323. Springer LNCS 3299, 2004.

[13] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proc. of the IEEE*, 77(4):541–580, 1989.

[14] J.L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[15] A. Rakow. Slicing Petri Nets. Technical report, Department für Informatik, Carl von Ossietzky Universität, Oldenburg, 2007.

[16] A. Rakow. Slicing Petri Nets with an Application to Workflow Verification. In *Proc. of the 34th Conf. on Current Trends in Theory and Practice of Computer Science (SOFSEM 2008)*, pages 436–447. Springer LNCS 4910, 2008.

[17] M. Rauhamaa. *A Comparative Study of Methods for Efficient Reachability Analysis*. Licentiate's thesis, Helsinki University of Technology, Department of Computer Science and Engineering, Digital Systems Laboratory, 1990.

[18] A.M. Sloane and J. Holdsworth. Beyond traditional program slicing. In *Proc. of the Int'l Symp. on Software Testing and Analysis*, pages 180–186, San Diego, CA, 1996. ACM Press.

[19] F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[20] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.