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.

Keywords: 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

[☆]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)

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.

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.

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 ($a \to P$) 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., $X = \emptyset$)

$$
\begin{array}{ll}
\textit{Domains} & \\
\quad M, N \ldots \in \mathit{Names} & \text{(Process names)} \\
\quad P, Q \ldots \in \mathit{Procs} & \text{(Processes)} \\
\quad a, b \ldots \in \Sigma & \text{(Events)}
\end{array}
$$

$$
\begin{array}{lcll}
S & ::= & D_1 \ldots D_m & \text{(Entire specification)} \\
D & ::= & N = P & \text{(Process definition)} \\
P & ::= & M & \text{(Process call)} \\
 & \mid & a \to P & \text{(Prefixing)} \\
 & \mid & P \sqcap Q & \text{(Internal choice)} \\
 & \mid & P \,\Box\, Q & \text{(External choice)} \\
 & \mid & P \underset{X}{\parallel} Q & \text{(Synchronized parallelism)} \qquad X \subseteq \Sigma \\
 & \mid & \mathit{STOP} & \text{(Stop)}
\end{array}
$$

Figure 1: Syntax of CSP specifications

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.*

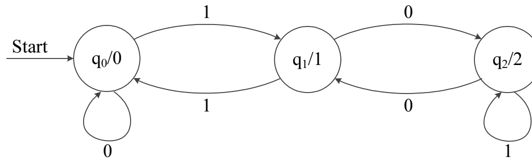


Figure 2: Moore machine to determine the remainder of a binary number divided by three

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`*).*

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

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.*

$$
\begin{aligned}
&\texttt{MAIN} = \texttt{REM0} \underset{\{\texttt{0},\texttt{1},\texttt{divisible3}\}}{\|} \texttt{BINARY}\\
&\texttt{REM0} = (\texttt{0} \to \texttt{REM0}) \,\square\, (\texttt{1} \to \texttt{REM1}) \,\square\, (\texttt{divisible3} \to \texttt{STOP})\\
&\texttt{REM1} = (\texttt{0} \to \texttt{REM2}) \,\square\, (\texttt{1} \to \texttt{REM0})\\
&\texttt{REM2} = (\texttt{0} \to \texttt{REM1}) \,\square\, (\texttt{1} \to \texttt{REM2})\\
&\texttt{BINARY} = \texttt{1} \to \texttt{1} \to \texttt{0} \to \texttt{divisible3} \to \texttt{STOP}
\end{aligned}
$$

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.

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.

(Process Call)
$$\frac{}{N \stackrel{\tau}{\longrightarrow} rhs(N)}$$

(Prefixing)
$$\frac{}{(a \rightarrow P) \stackrel{a}{\longrightarrow} P}$$

(Internal Choice 1)
$$\frac{}{(P \sqcap Q) \stackrel{\tau}{\longrightarrow} P}$$

(Internal Choice 2)
$$\frac{}{(P \sqcap Q) \stackrel{\tau}{\longrightarrow} Q}$$

(External Choice 1)
$$\frac{P \stackrel{\tau}{\longrightarrow} P'}{(P \;\Box\; Q) \stackrel{\tau}{\longrightarrow} (P' \;\Box\; Q)}$$

(External Choice 2)
$$\frac{Q \stackrel{\tau}{\longrightarrow} Q'}{(P \;\Box\; Q) \stackrel{\tau}{\longrightarrow} (P \;\Box\; Q')}$$

(External Choice 3)
$$\frac{P \stackrel{e}{\rightarrow} P'}{(P \;\Box\; Q) \stackrel{e}{\rightarrow} P'}\ e \in \Sigma$$

(External Choice 4)
$$\frac{Q \stackrel{e}{\rightarrow} Q'}{(P \;\Box\; Q) \stackrel{e}{\rightarrow} Q'}\ e \in \Sigma$$

(Synchronized Parallelism 1)
$$\frac{P \stackrel{e}{\rightarrow} P'}{(P \underset{X}{||} Q) \stackrel{e}{\rightarrow} (P' \underset{X}{||} Q)}\ e \in \Sigma^\tau \backslash X$$

(Synchronized Parallelism 2)
$$\frac{Q \stackrel{e}{\rightarrow} Q'}{(P \underset{X}{||} Q) \stackrel{e}{\rightarrow} (P \underset{X}{||} Q')}\ e \in \Sigma^\tau \backslash X$$

(Synchronized Parallelism 3)
$$\frac{P \stackrel{e}{\longrightarrow} P' \quad Q \stackrel{e}{\longrightarrow} Q'}{(P \underset{X}{||} Q) \stackrel{e}{\longrightarrow} (P' \underset{X}{||} Q')}\ e \in X$$

Figure 3: 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 $e \in \Sigma$ 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* $\texttt{STOP} \underset{\{\texttt{0,1,divisible3}\}}{||} \texttt{STOP}$*. This computation corresponds to the case in which the binary number 110, divisible by 3, produces the sequence of events* $\langle \texttt{1}, \texttt{1}, \texttt{0}, \texttt{divisible3} \rangle$*. 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:

$$traces(\texttt{MAIN}) = \{\langle(\texttt{0}|\texttt{1})^*\rangle, \langle(\texttt{0}^*|\texttt{1}(\texttt{0}\texttt{1}^*\texttt{0})^*\texttt{1})^*\texttt{divisible3}\rangle\} \quad (1)$$

The set of all traces of process MAIN in the CSP specification of Example 2 is:

$$traces(\texttt{MAIN}) = \{\langle\rangle, \langle\texttt{1}\rangle, \langle\texttt{1},\texttt{1}\rangle, \langle\texttt{1},\texttt{1},\texttt{0}\rangle, \langle\texttt{1},\texttt{1},\texttt{0},\texttt{divisible3}\rangle\} \quad (2)$$

$$(\textsf{Process Call})\frac{}{\texttt{MAIN} \xrightarrow{\tau} (\texttt{REM0} \underset{\{\texttt{0,1,divisible3}\}}{\parallel} \texttt{BINARY})}$$

$$\begin{array}{c}(\textsf{Syncronized}\\ \textsf{Parallelism 1})\end{array}\frac{(\textsf{Process Call})\dfrac{}{\texttt{REM0} \xrightarrow{\tau} (((\texttt{0} \to \texttt{REM0}) \,\Box\, (\texttt{1} \to \texttt{REM1})) \,\Box\, (\texttt{divisible3} \to \texttt{STOP}))}}{(\texttt{REM0} \underset{\{\texttt{0,1,divisible3}\}}{\parallel} \texttt{BINARY}) \xrightarrow{\tau} State_1} \text{ where}$$

$$State_1 = (((\texttt{0} \to \texttt{REM0}) \,\Box\, (\texttt{1} \to \texttt{REM1})) \,\Box\, (\texttt{divisible3} \to \texttt{STOP})) \underset{\{\texttt{0,1,divisible3}\}}{\parallel} \texttt{BINARY}$$

$$\begin{array}{c}(\textsf{Syncronized}\\ \textsf{Parallelism 2})\end{array}\frac{(\textsf{Process Call})\dfrac{}{\texttt{BINARY} \xrightarrow{\tau} (\texttt{1} \to \texttt{1} \to \texttt{0} \to \texttt{divisible3} \to \texttt{STOP})}}{State_1 \xrightarrow{\tau} State_2} \text{ where}$$

$$State_2 = (((\texttt{0} \to \texttt{REM0})\Box(\texttt{1} \to \texttt{REM1}))\Box(\texttt{divisible3} \to \texttt{STOP})) \underset{\{\texttt{0,1,divisible3}\}}{\parallel} (\texttt{1} \to \texttt{1} \to \texttt{0} \to \texttt{divisible3} \to \texttt{STOP})$$

$$\begin{array}{c}(\textsf{Syncronized}\\ \textsf{Parallelism 3})\end{array}\frac{Left \quad Right}{State_2 \xrightarrow{1} State_3} \text{ where}$$

$$Left = (\textsf{External Choice 3})\frac{(\textsf{External Choice 4})\dfrac{(\textsf{Prefixing})\dfrac{}{(\texttt{1} \to \texttt{REM1}) \xrightarrow{1} \texttt{REM1}}}{((\texttt{0} \to \texttt{REM0})\Box(\texttt{1} \to \texttt{REM1})) \xrightarrow{1} \texttt{REM1}}}{(((\texttt{0} \to \texttt{REM0})\Box(\texttt{1} \to \texttt{REM1}))\Box(\texttt{divisible3} \to \texttt{STOP})) \xrightarrow{1} \texttt{REM1}}$$

$$Right = (\textsf{Prefixing})\frac{}{(\texttt{1} \to \texttt{1} \to \texttt{0} \to \texttt{divisible3} \to \texttt{STOP}) \xrightarrow{1} (\texttt{1} \to \texttt{0} \to \texttt{divisible3} \to \texttt{STOP})}$$

$$\text{and } State_3 = \texttt{REM1} \underset{\{\texttt{0,1,divisible3}\}}{\parallel} (\texttt{1} \to \texttt{0} \to \texttt{divisible3} \to \texttt{STOP})$$

$$\begin{array}{c}(\textsf{Syncronized}\\ \textsf{Parallelism 1})\end{array}\frac{(\textsf{Process Call})\dfrac{}{\texttt{REM1} \xrightarrow{\tau} ((\texttt{0} \to \texttt{REM2}) \,\Box\, (\texttt{1} \to \texttt{REM0}))}}{State_3 \xrightarrow{\tau} State_4} \text{ where}$$

$$State_4 = ((\texttt{0} \to \texttt{REM2}) \,\Box\, (\texttt{1} \to \texttt{REM0})) \underset{\{\texttt{0,1,divisible3}\}}{\parallel} (\texttt{1} \to \texttt{0} \to \texttt{divisible3} \to \texttt{STOP})$$

$$\begin{array}{c}(\textsf{Syncronized}\\ \textsf{Parallelism 3})\end{array}\frac{Left \quad Right}{State_4 \xrightarrow{1} State_5} \text{ where}$$

$$Left = (\textsf{External Choice 4})\frac{(\textsf{Prefixing})\dfrac{}{(\texttt{1} \to \texttt{REM0}) \xrightarrow{1} \texttt{REM0}}}{((\texttt{0} \to \texttt{REM2}) \,\Box\, (\texttt{1} \to \texttt{REM0})) \xrightarrow{1} \texttt{REM0}}$$

$$Right = (\textsf{Prefixing})\frac{}{(\texttt{1} \to \texttt{0} \to \texttt{divisible3} \to \texttt{STOP}) \xrightarrow{1} (\texttt{0} \to \texttt{divisible3} \to \texttt{STOP})}$$

$$\text{and } State_5 = \texttt{REM0} \underset{\{\texttt{0,1,divisible3}\}}{\parallel} (\texttt{0} \to \texttt{divisible3} \to \texttt{STOP})$$

$$\begin{array}{c}(\textsf{Syncronized}\\ \textsf{Parallelism 1})\end{array}\frac{(\textsf{Process Call})\dfrac{}{\texttt{REM0} \xrightarrow{\tau} (((\texttt{0} \to \texttt{REM0}) \,\Box\, (\texttt{1} \to \texttt{REM1})) \,\Box\, (\texttt{divisible3} \to \texttt{STOP}))}}{State_5 \xrightarrow{\tau} State_6} \text{ where}$$

$$State_6 = (((\texttt{0} \to \texttt{REM0}) \,\Box\, (\texttt{1} \to \texttt{REM1})) \,\Box\, (\texttt{divisible3} \to \texttt{STOP})) \underset{\{\texttt{0,1,divisible3}\}}{\parallel} (\texttt{0} \to \texttt{divisible3} \to \texttt{STOP})$$

Figure 4: A computation with the operational semantics in Figure 3

$$\text{(Syncronized Parallelism 3)}\ \frac{Left \quad Right}{State_6 \xrightarrow{1} State_7} \text{ where}$$

$$Left = \text{(External Choice 3)}\ \frac{\text{(External Choice 3)}\ \frac{\text{(Prefixing)}\ \frac{}{(0 \rightarrow \texttt{REM0}) \xrightarrow{0} \texttt{REM0}}}{((0 \rightarrow \texttt{REM0})\Box(1 \rightarrow \texttt{REM1})) \xrightarrow{0} \texttt{REM0}}}{(((0 \rightarrow \texttt{REM0})\Box(1 \rightarrow \texttt{REM1}))\Box(\texttt{divisible3} \rightarrow \texttt{STOP})) \xrightarrow{0} \texttt{REM0}}$$

$$Right = \text{(Prefixing)}\ \frac{}{(0 \rightarrow \texttt{divisible3} \rightarrow \texttt{STOP}) \xrightarrow{0} (\texttt{divisible3} \rightarrow \texttt{STOP})}$$

$$\text{and } State_7 = \texttt{REM0} \underset{\{0,1,\texttt{divisible3}\}}{\parallel} (\texttt{divisible3} \rightarrow \texttt{STOP})$$

$$\text{(Syncronized Parallelism 1)}\ \frac{\text{(Process Call)}\ \frac{}{\texttt{REM0} \xrightarrow{\tau} (((0 \rightarrow \texttt{REM0}) \ \Box\ (1 \rightarrow \texttt{REM1})) \ \Box\ (\texttt{divisible3} \rightarrow \texttt{STOP}))}}{State_7 \xrightarrow{\tau} State_8} \text{ where}$$

$$State_8 = (((0 \rightarrow \texttt{REM0}) \ \Box\ (1 \rightarrow \texttt{REM1})) \ \Box\ (\texttt{divisible3} \rightarrow \texttt{STOP})) \underset{\{0,1,\texttt{divisible3}\}}{\parallel} (\texttt{divisible3} \rightarrow \texttt{STOP})$$

$$\text{(Syncronized Parallelism 3)}\ \frac{Left \quad Right}{State_8 \xrightarrow{\texttt{divisible3}} \texttt{STOP} \underset{\{0,1,\texttt{divisible3}\}}{\parallel} \texttt{STOP}} \text{ where}$$

$$Left = \text{(External Choice 4)}\ \frac{\text{(Prefixing)}\ \frac{}{(\texttt{divisible3} \rightarrow \texttt{STOP}) \xrightarrow{\texttt{divisible3}} \texttt{STOP}}}{(((0 \rightarrow \texttt{REM0})\Box(1 \rightarrow \texttt{REM1}))\Box(\texttt{divisible3} \rightarrow \texttt{STOP})) \xrightarrow{\texttt{divisible3}} \texttt{STOP}}$$

$$Right = \text{(Prefixing)}\ \frac{}{(\texttt{divisible3} \rightarrow \texttt{STOP}) \xrightarrow{\texttt{divisible3}} \texttt{STOP}}$$

Figure 4: A computation with the operational semantics in Figure 3 (cont.)

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 $F : P \times T \ \cup\ T \times P \rightarrow \mathbb{N}$*. A* marking *of a Petri net is a function $M : P \rightarrow \mathbb{N}$. 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 $\mathcal{L}_P : P \rightarrow \mathcal{P}$ *and $\mathcal{L}_T$ is a* labelling function $\mathcal{L}_T : T \rightarrow \mathcal{T}$.

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).

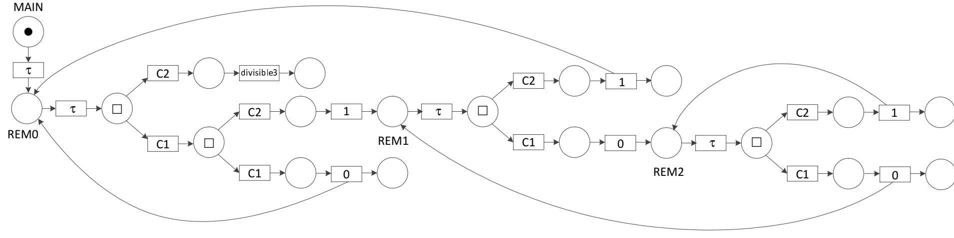


Figure 5: Petri net associated with the specification of Example 1

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 $t^\bullet$ 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,*

$$
\begin{aligned}
{}^\bullet t = \{p \in P \mid (p, t) \in F\} \text{ and } t^\bullet = \{p \in P \mid (t, p) \in F\} \\
{}^\bullet p = \{t \in T \mid (t, p) \in F\} \text{ and } p^\bullet = \{t \in T \mid (p, t) \in F\}
\end{aligned}
$$

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 *$\sigma = t_1 t_2 \ldots t_n$ such that $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} M_n$. 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:*

$$\begin{aligned} L_{\mathcal{A}}(\mathcal{N}) = \{\langle\rangle\} \cup \{\langle \mathcal{L}_T(s_1), \ldots, \mathcal{L}_T(s_m)\rangle \mid \exists \sigma = t_1 \ldots t_n \in FS(\mathcal{N}) \wedge \\ \forall i, 1 \le i \le m \textit{ with } m \le n : s_i = t_k \textit{ with } 1 \le k \le n \wedge \\ \mathcal{L}_T(t_k) \in \mathcal{A} \wedge \nexists j, 1 \le j < i : s_j = t_l \textit{ with } l \ge k \ \} \end{aligned}$$

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:

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

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 $\mathcal{N}_0 = (\langle\{p_0\}, \emptyset, \emptyset\rangle, M_0, \mathcal{P}, \mathcal{T}, \mathcal{L}_P, \mathcal{L}_T)$, $M_0(p_0) = 1$,
$\mathcal{P} = Names \cup \{\Box, \sqcap\}$, $\mathcal{T} = \Sigma^\tau \cup \{\|, \texttt{C1}, \texttt{C2}\}$.
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 $a \in \mathcal{T}$.

(Process Call - Sequential)

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

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

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

(Process Call - Parallel)

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

(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)

$$\frac{}{(P \boxminus Q, p, \mathcal{N}, (S, S_0), _) \xrightarrow{\tau} (P', p', \mathcal{N}', (S', S_0'), \emptyset)} \quad \boxminus \in \{\sqcap, \sqcup\}$$

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

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

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

(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)

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

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

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

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

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

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

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

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

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

Figure 6: An instrumented operational semantics that generates a Petri net (cont.)

$P1$ 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 $t_\parallel$. This 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 $\circlearrowleft$, 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`. 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 $\circlearrowleft$), 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 $\mathcal{N}$ 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 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 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

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

(Synchronized Parallelism 4)

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

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

$$\texttt{LoopControl}(P1 \underset{X}{\parallel}_{(p_1,p_2,\Upsilon)} P2, \mathcal{N}) =$$
$$\begin{cases} (\circlearrowright(P1'' \underset{X}{\parallel}_{(p_1',p_2',\Upsilon)} P2''), \mathcal{N}) & \text{if } P1' = \circlearrowright(P1'') \wedge (P2' = \circlearrowright(P2'')\vee \\ & \quad ((P2' = \texttt{STOP} \vee (P1''\|P2', _, _) \in \Upsilon) \;\wedge\; P2'' = P2')) \\ (P1''' \underset{X}{\parallel}_{(p_1',p_2',\Upsilon)} P2', \mathcal{N}') & \text{if } P1' = \circlearrowright(P1'') \wedge P2' \neq \circlearrowright(_) \wedge P2' \neq \texttt{STOP} \wedge \\ & \quad (P1''\|P2', _, _) \notin \Upsilon \;\wedge\; (P1''', \mathcal{N}') = \texttt{DelEdges}(P1'', \mathcal{N}) \\ (\texttt{STOP}, \mathcal{N}) & \text{otherwise} \end{cases}$$
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 6: An instrumented operational semantics that generates a Petri net (cont.)

$\circlearrowright$; 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 $\circlearrowright$ 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 $P \underset{X}{\parallel} Q$.

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.

(Synchronized Parallelism 5)

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

$$\begin{array}{l}
S = [] \;\vee\; ((S = (_ : (\mathsf{SP1}, _)) \;\vee\; S = (_ : (\mathsf{SP2}, _)) \;\vee\; S = (_ : (\mathsf{SP3}, _))) \\
\qquad\qquad \wedge\, (P1 \underset{X}{\|} P2, _, _) \in \Upsilon) \\
\wedge (P, \mathcal{N}_P, S_P) = \texttt{CheckLoops}(P1 \underset{X}{\|}_{(p1,p2,\Upsilon)} P2, \mathcal{N}) \\
\wedge ((S = [] \wedge S_P = [] \wedge e = \tau \wedge (P'', \mathcal{N}'', (S'', S''_0), \Delta') = (P, \mathcal{N}_P, ([], S_0), \emptyset)) \\
\quad \vee ((S = [] \wedge S_P \neq [] \wedge S'_P = S_P) \vee (S \neq [] \wedge S'_P = S) \\
\qquad \wedge (P'', \mathcal{N}'', (S'', S''_0), \Delta') = (P', \mathcal{N}', (S', S'_0), \Delta)))
\end{array}$$

$$\texttt{CheckLoops}(P1 \underset{X}{\|}_{(p1,p2,\Upsilon)} P2, \mathcal{N}) =$$

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

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

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

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

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

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

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

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

Figure 6: An instrumented operational semantics that generates a Petri net (cont.)

Example 4. *Consider again the specification of Example 2. Due to the set of synchronized events* $\{\mathtt{0},\mathtt{1},\mathtt{divisible3}\}$ *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* $traces(\mathtt{MAIN})$ *in (2). In particular, it can produce the sequence of events* $\langle \mathtt{1},\mathtt{1},\mathtt{0},\mathtt{divisible3}\rangle$ *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.*

First iteration

$State_0 = (\mathtt{MAIN}, p_0, \mathcal{N}_0, ([],[]), \emptyset)$ (PC-Seq)

where $\mathcal{N}_0 = (\langle\{p_0\},\emptyset,\emptyset\rangle, M_0, \mathcal{P}, \mathcal{T}, \mathcal{L}_P, \mathcal{L}_T)$, $M_0(p_0) = 1$,

$\mathcal{P} = Names \cup \{\Box, \sqcap, \mathtt{STOP}\}$, $\mathcal{T} = \Sigma^\tau \cup \{\|, \mathtt{C1}, \mathtt{C2}\}$

$State_1 = (\mathtt{REM0}_\diamond \underset{\{\mathtt{0,1,divisible3}\}}{\|}_{(\bot,\bot,\emptyset)} \mathtt{BINARY}_\diamond, p_1, \mathcal{N}_1, ([],[]), \emptyset)$ (SP5)(SP1) (PC-Par)

where $\mathcal{N}_1 = \mathcal{N}_0[p_{\mathtt{MAIN}} \mapsto t_\tau \mapsto p_1]$ and $\mathcal{L}_P(p_0) = \mathtt{MAIN}$

$State_2 = (rhs(\mathtt{REM0}) \underset{\{\mathtt{0,1,divisible3}\}}{\|}_{lab_2} \mathtt{BINARY}_\diamond, p_1, \mathcal{N}_2, ([], S_2), \emptyset)$ (SP5)(SP2) (PC-Par)

where $rhs(\mathtt{REM0}) = (((\mathtt{0} \rightarrow \mathtt{REM0}) \Box (\mathtt{1} \rightarrow \mathtt{REM1})) \Box (\mathtt{divisible3} \rightarrow \mathtt{STOP}))$,

$\mathcal{N}_2 = \mathcal{N}_1[p_1 \mapsto t_\| \mapsto p_2, p_1 \mapsto t_\| \mapsto p_3, p_{\mathtt{REM0}} \mapsto t_\tau \mapsto p_4]$, $\mathcal{L}_P(p_2) = \mathtt{REM0}$,

$lab_2 = (p_4, p_3, \Upsilon_2)$, $\Upsilon_2 = \{(\mathtt{REM0}\|\mathtt{BINARY}, p_2, p_3)\}$ and $S_2 = [(\mathtt{SP1}, \emptyset)]$

$State_3 = (rhs(\mathtt{REM0}) \underset{\{\mathtt{0,1,divisible3}\}}{\|}_{lab_3} rhs(\mathtt{BINARY}), p_1, \mathcal{N}_3, ([], S_3), \emptyset)$ (SP5)(SP1) (Choice)

where $rhs(\mathtt{BINARY}) = (\mathtt{1} \rightarrow \mathtt{1} \rightarrow \mathtt{0} \rightarrow \mathtt{divisible3} \rightarrow \mathtt{STOP})$,

$\mathcal{N}_3 = \mathcal{N}_2[p_{\mathtt{BINARY}} \mapsto t_\tau \mapsto p_5]$, $\mathcal{L}_P(p_3) = \mathtt{BINARY}$, $lab_3 = (p_4, p_5, \Upsilon_3)$,

$\Upsilon_3 = \Upsilon_2 \cup \{(rhs(\mathtt{REM0})\|\mathtt{BINARY}, p_4, p_3)\}$ and $S_3 = (\mathtt{SP2}, \emptyset) : S_2$

$State_4 = (((\mathtt{0} \rightarrow \mathtt{REM0}) \Box (\mathtt{1} \rightarrow \mathtt{REM1})) \underset{\{\mathtt{0,1,divisible3}\}}{\|}_{lab_4} rhs(\mathtt{BINARY}), p_1, \mathcal{N}_4, ([], S_4), \emptyset)$ (SP5)(SP1) (Choice)

where $\mathcal{N}_4 = \mathcal{N}_3[p_\Box \mapsto t_{\mathtt{C1}} \mapsto p_6]$, $\mathcal{L}_P(p_4) = \Box$, $lab_4 = (p_6, p_5, \Upsilon_4)$,

$\Upsilon_4 = \Upsilon_3 \cup \{(rhs(\mathtt{REM0})\|rhs(\mathtt{BINARY}), p_4, p_5)\}$ and

$S_4 = [(\mathtt{C1}, \{\mathtt{C2}\}), (\mathtt{SP1}, \emptyset)] : S_3$

$State_5 = ((\mathtt{0} \rightarrow \mathtt{REM0}) \underset{\{\mathtt{0,1,divisible3}\}}{\|}_{lab_5} rhs(\mathtt{BINARY}), p_1, \mathcal{N}_5, ([], S_5), \emptyset)$ (SP5)(SP4)

where $\mathcal{N}_5 = \mathcal{N}_4[p_\Box \mapsto t_{\mathtt{C1}} \mapsto p_7]$, $\mathcal{L}_P(p_6) = \Box$, $lab_5 = (p_7, p_5, \Upsilon_5)$

$\Upsilon_5 = \Upsilon_4 \cup \{(((\mathtt{0} \rightarrow \mathtt{REM0}) \Box (\mathtt{1} \rightarrow \mathtt{REM1}))\|rhs(\mathtt{BINARY}), p_6, p_5)\}$ and

$S_5 = [(\mathtt{C1}, \{\mathtt{C2}\}), (\mathtt{SP1}, \emptyset)] : S_4$

$State_6 = (\mathtt{STOP}, p_1, \mathcal{N}_5, ([], S_6), \emptyset)$ where $S_6 = (\mathtt{SP4}, \emptyset) : S_5$

Figure 7: First iteration of Algorithm 1 for Example 2

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 $(\mathtt{MAIN}, p_0, \mathcal{N}_0, ([],[]), \emptyset)$ *and get the computation* ***First iteration****. This computation corresponds to the execution of the left branch of the two choices*

of process REMO*. 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:

$$
\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},\ \texttt{1}\rangle,\\
& \langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ \texttt{1},\ \tau\rangle, \langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ \texttt{1},\ \tau,\ \texttt{C1}\rangle,\\
& \langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ \texttt{1},\ \tau,\ \texttt{C2}\rangle, \langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ \texttt{1},\ \tau,\ \texttt{C2},\ \texttt{1}\rangle,\\
& \langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ \texttt{1},\ \tau,\ \texttt{C2},\ \texttt{1},\ \tau\rangle,\\
& \langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ \texttt{1},\ \tau,\ \texttt{C2},\ \texttt{1},\ \tau,\ \texttt{C2}\rangle,\\
& \langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ \texttt{1},\ \tau,\ \texttt{C2},\ \texttt{1},\ \tau,\ \texttt{C1}\rangle,\\
& \langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ \texttt{1},\ \tau,\ \texttt{C2},\ \texttt{1},\ \tau,\ \texttt{C1},\ \texttt{C2}\rangle,\\
& \langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ \texttt{1},\ \tau,\ \texttt{C2},\ \texttt{1},\ \tau,\ \texttt{C1},\ \texttt{C1}\rangle,\\
& \langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ \texttt{1},\ \tau,\ \texttt{C2},\ \texttt{1},\ \tau,\ \texttt{C1},\ \texttt{C1},\ \texttt{0}\rangle,\\
& \langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ \texttt{1},\ \tau,\ \texttt{C2},\ \texttt{1},\ \tau,\ \texttt{C1},\ \texttt{C1},\ \texttt{0},\ \tau\rangle,\\
& \langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ \texttt{1},\ \tau,\ \texttt{C2},\ \texttt{1},\ \tau,\ \texttt{C1},\ \texttt{C1},\ \texttt{0},\ \tau,\ \texttt{C1}\rangle,\\
& \langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ \texttt{1},\ \tau,\ \texttt{C2},\ \texttt{1},\ \tau,\ \texttt{C1},\ \texttt{C1},\ \texttt{0},\ \tau,\ \texttt{C1},\ \texttt{C1}\rangle,\\
& \langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ \texttt{1},\ \tau,\ \texttt{C2},\ \texttt{1},\ \tau,\ \texttt{C1},\ \texttt{C1},\ \texttt{0},\ \tau,\ \texttt{C1},\ \texttt{C2}\rangle,\\
& \langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ \texttt{1},\ \tau,\ \texttt{C2},\ \texttt{1},\ \tau,\ \texttt{C1},\ \texttt{C1},\ \texttt{0},\ \tau,\ \texttt{C2}\rangle,\\
& \langle\tau,\ ||,\ \tau,\ \tau,\ \texttt{C1},\ \texttt{C2},\ \texttt{1},\ \tau,\ \texttt{C2},\ \texttt{1},\ \tau,\ \texttt{C1},\ \texttt{C1},\ \texttt{0},\ \tau,\ \texttt{C2},\ \texttt{divisible3}\rangle\}
\end{aligned}
$$

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

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

We can see that this language is exactly the same as the one produced by traces(MAIN) *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

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 $S_7 = [(\texttt{C2}, \emptyset), (\texttt{SP1}, \emptyset), (\texttt{C1}, \{\texttt{C2}\}), (\texttt{SP1}, \emptyset), (\texttt{SP2}, \emptyset), (\texttt{SP1}, \emptyset)]$

$State_8 = (\texttt{REM0}_\diamond \underset{\{\texttt{0,1,divisible3}\}}{\parallel} {}_{(\perp,\perp,\emptyset)}\texttt{BINARY}_\diamond, p_1, \mathcal{N}_5, (S_7, []), \emptyset)$ where $\mathcal{L}_P(p_0) = \texttt{MAIN}$ (SP1)(PC-Par)

$State_9 = (rhs(\texttt{REM0}) \underset{\{\texttt{0,1,divisible3}\}}{\parallel} {}_{lab_9}\texttt{BINARY}_\diamond, p_1, \mathcal{N}_5, (S_8, [(\texttt{SP1}, \emptyset)]), \emptyset)$ (SP2)(PC-Par)

where $rhs(\texttt{REM0}) = (((\texttt{0} \rightarrow \texttt{REM0}) \Box (\texttt{1} \rightarrow \texttt{REM1})) \Box (\texttt{divisible3} \rightarrow \texttt{STOP}))$,

$\mathcal{L}_P(p_2) = \texttt{REM0}, lab_9 = (p_4, p_3, \Upsilon_9), \Upsilon_9 = \{(\texttt{REM0} \| \texttt{BINARY}, p_2, p_3)\}$ and

$S_8 = [(\texttt{C2}, \emptyset), (\texttt{SP1}, \emptyset), (\texttt{C1}, \{\texttt{C2}\}), (\texttt{SP1}, \emptyset), (\texttt{SP2}, \emptyset)]$

$State_{10} = (rhs(\texttt{REM0}) \underset{\{\texttt{0,1,divisible3}\}}{\parallel} {}_{lab_{10}} rhs(\texttt{BINARY}), p_1, \mathcal{N}_5, (S_9, S_{10}), \emptyset)$ (SP1)(Choice)

where $rhs(\texttt{BINARY}) = (\texttt{1} \rightarrow \texttt{1} \rightarrow \texttt{0} \rightarrow \texttt{divisible3} \rightarrow \texttt{STOP}), \mathcal{L}_P(p_3) = \texttt{BINARY}$,

$lab_{10} = (p_4, p_5, \Upsilon_{10}), \Upsilon_{10} = \Upsilon_9 \cup \{(rhs(\texttt{REM0}) \| \texttt{BINARY}, p_4, p_3)\}$,

$S_9 = [(\texttt{C2}, \emptyset), (\texttt{SP1}, \emptyset), (\texttt{C1}, \{\texttt{C2}\}), (\texttt{SP1}, \emptyset)]$ and $S_{10} = [(\texttt{SP2}, \emptyset) : (\texttt{SP1}, \emptyset)]$

$State_{11} = (((\texttt{0} \rightarrow \texttt{REM0}) \Box (\texttt{1} \rightarrow \texttt{REM1})) \underset{\{\texttt{0,1,divisible3}\}}{\parallel} {}_{lab_{11}} rhs(\texttt{BINARY}), p_1, \mathcal{N}_5, (S_{11}, S_{12}), \emptyset)$ (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 $S_{12} = [(\texttt{C1}, \{\texttt{C2}\}), (\texttt{SP1}, \emptyset)] : S_{10}$

$State_{12} = ((\texttt{1} \rightarrow \texttt{REM1}) \underset{\{\texttt{0,1,divisible3}\}}{\parallel} {}_{lab_{12}} rhs(\texttt{BINARY}), p_1, \mathcal{N}_6, ([], S_{13}), \emptyset)$ (SP5)(SP3)(Pref)(Pref)

where $\mathcal{N}_6 = \mathcal{N}_5[p_\Box \mapsto t_{\texttt{C2}} \mapsto p_8], \mathcal{L}_P(p_6) = \Box, lab_{12} = (p_8, p_5, \Upsilon_{12})$,

$\Upsilon_{12} = \Upsilon_{11} \cup \{(((\texttt{0} \rightarrow \texttt{REM0}) \Box (\texttt{1} \rightarrow \texttt{REM1})) \| rhs(\texttt{BINARY}), p_6, p_5)\}$ and

$S_{13} = [(\texttt{C2}, \emptyset), (\texttt{SP1}, \emptyset)] : S_{12}$

$State_{13} = (\texttt{REM1}_\diamond \underset{\{\texttt{0,1,divisible3}\}}{\parallel} {}_{lab_{13}} (\texttt{1} \rightarrow \texttt{0} \rightarrow \texttt{divisible3} \rightarrow \texttt{STOP}), p_1, \mathcal{N}_7, ([], S_{14}), \emptyset)$ (SP5)(SP1)(PC-Par)

where $\mathcal{N}_7 = \mathcal{N}_6[p_8 \mapsto t_1 \mapsto p_9, p_5 \mapsto t_1 \mapsto p_{10}], lab_{13} = (p_9, p_{10}, \Upsilon_{13})$,

$\Upsilon_{13} = \Upsilon_{12} \cup \{((\texttt{1} \rightarrow \texttt{REM1}) \| rhs(\texttt{BINARY}), p_8, p_5)\}$

$S_{14} = (\texttt{SP3}, \emptyset) : S_{13}$ and $\Delta_1 = \{(p_8, t_1, p_9), (p_5, t_1, p_{10})\}$

$State_{14} = (rhs(\texttt{REM1}) \underset{\{\texttt{0,1,divisible3}\}}{\parallel} {}_{lab_{14}} (\texttt{1} \rightarrow \texttt{0} \rightarrow \texttt{divisible3} \rightarrow \texttt{STOP}), p_1, \mathcal{N}_8, ([], S_{15}), \emptyset)$ (SP5)(SP1)(Choice)

where $rhs(\texttt{REM1}) = ((\texttt{0} \rightarrow \texttt{REM2}) \Box (\texttt{1} \rightarrow \texttt{REM0})), \mathcal{N}_8 = \mathcal{N}_7[p_{\texttt{REM1}} \mapsto t_\tau \mapsto p_{11}]$,

$\mathcal{L}_P(p_9) = \texttt{REM1}, lab_{14} = (p_{11}, p_{10}, \Upsilon_{14})$

$\Upsilon_{14} = \Upsilon_{13} \cup \{(\texttt{REM1} \| rhs(\texttt{BINARY}), p_9, p_{10})\}$ and $S_{15} = (\texttt{SP1}, \emptyset) : S_{14}$

$State_{15} = ((\texttt{0} \rightarrow \texttt{REM2}) \underset{\{\texttt{0,1,divisible3}\}}{\parallel} {}_{lab_{15}} (\texttt{1} \rightarrow \texttt{0} \rightarrow \texttt{divisible3} \rightarrow \texttt{STOP}), p_1, \mathcal{N}_9, ([], S_{16}), \emptyset)$ (SP5)(SP4)

where $\mathcal{N}_9 = \mathcal{N}_8[p_\Box \mapsto t_{\texttt{C1}} \mapsto p_{12}], \mathcal{L}_P(p_{11}) = \Box, lab_{15} = (p_{12}, p_{10}, \Upsilon_{15})$

$\Upsilon_{15} = \Upsilon_{14} \cup \{(rhs(\texttt{REM1}) \| (\texttt{1} \rightarrow \texttt{0} \rightarrow \texttt{divisible3} \rightarrow \texttt{STOP}), p_{11}, p_{10})\}$ 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

step the sequence of events (both internal and external) that happened during

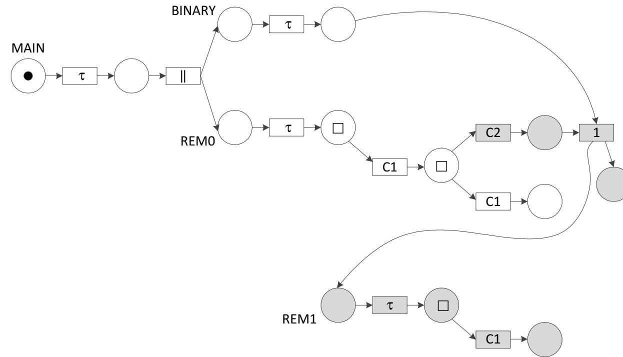
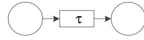


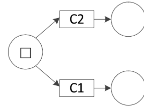
Figure 9: Petri net generated in the **First** and **Second iterations** of Algorithm 1 for Example 2

the evaluation of the semantics. For instance,

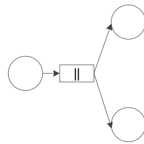
- Each occurrence of a τ is explicitly represented in the Petri net with a transition (labeled with the internal event τ) between two places:



- 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:



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

(a) PN generated with Algorithm 1

(b) PN optimized

Figure 10: PN associated with the specification of Example 2

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).

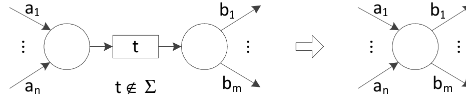
The task of deleting duplicate nodes is performed by function `DelDuplicates` whose implementation can be found in Algorithm 3. The task of deleting use-

less 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.

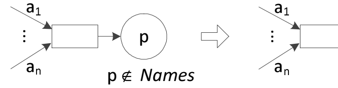
Function `DelDuplicates` traverses the Petri net from the initial place (labeled with `MAIN`) following all paths. During the traversal, it tries to identify repeated parts of the Petri net to remove the repetitions and reuse one of them, whenever it is possible. For this, three auxiliary functions implemented 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:

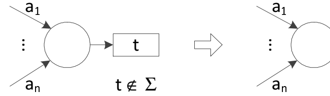
- Removing useless transitions:



- 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`$((\langle P, T, F \rangle, M_0, \mathcal{P}, \mathcal{T}, \mathcal{L}_P, \mathcal{L}_T))$
 $Visited = \emptyset$, $Pending = \{p_0 \in P \mid \mathcal{L}_P(p_0) = \texttt{MAIN}\}$
 foreach $n \in Pending$
  if $n \in ((P \cup T) \setminus Visited)$
  then $Eqs = \{n_{eq} \in P \cup T \mid n_{eq} \neq n \wedge \texttt{Equal}(n_{eq}, n, P, T, F, \mathcal{L}_P, \mathcal{L}_T)\}$
   if $Eqs = \emptyset$
   then $Pending = Pending \cup (n^\bullet \setminus Visited)$
   else if $n \in P$
    then foreach $n_{eq} \in Eqs$
     $F = (F \cup \{(n_p, n) \mid n_p \in {}^\bullet n_{eq}\}) \setminus \{(n_p, n_{eq}) \in F\}$
     $(P, T, F) =$ `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`$(t_{eq}, t_{eq}^\bullet, P, T, F)$
        $F = F \cup \{(t_{eq}, p_{new})\}$
    $Visited = Visited \cup Eqs$
  $Pending = Pending \setminus \{n\}$
  $Visited = Visited \cup \{n\}$

 return $(P, T, F, \mathcal{L}_T)$

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

$$\begin{cases} \mathsf{true} & \text{if } (n \in P \wedge n = n') \vee \\ & \quad ((n^\bullet \cup n'^\bullet) = \emptyset \wedge \texttt{SameLbl}(n, n') \wedge \\ & \quad \texttt{Comparable}(n) \wedge \texttt{Comparable}(n')) \\ \texttt{Eq}(n_s, n'_s) & \text{if } \texttt{SameLbl}(n, n') \wedge \\ & \quad \texttt{Comparable}(n) \wedge \texttt{Comparable}(n') \wedge \\ & \quad n^\bullet = \{n_s\} \wedge n'^\bullet = \{n'_s\} \\ (\texttt{Eq}(n_{s1}, n'_{s1}) \wedge \texttt{Eq}(n_{s2}, n'_{s2})) \vee & \text{if } \texttt{SameLbl}(n, n') \wedge \\ (\texttt{Eq}(n_{s1}, n'_{s2}) \wedge \texttt{Eq}(n_{s2}, n'_{s1})) & \quad \texttt{Comparable}(n) \wedge \texttt{Comparable}(n') \wedge \\ & \quad n^\bullet = \{n_{s1}, n_{s2}\} \wedge n'^\bullet = \{n'_{s1}, n'_{s2}\} \\ \mathsf{false} & \text{otherwise} \end{cases}$$

where $\texttt{Eq}(n, n') = \texttt{Equal}(n, n', P, T, F, \mathcal{L}_P, \mathcal{L}_T)$
  $\texttt{SameLbl}(n, n') = (\nexists \mathcal{L}_P(n) \wedge \nexists \mathcal{L}_P(n')) \vee \mathcal{L}_T(n) = \mathcal{L}_T(n') \vee \mathcal{L}_P(n) = \mathcal{L}_P(n')$
  $\texttt{Comparable}(n) = (n \in P \wedge n^\bullet = \emptyset) \;\vee\; {}^\bullet n = \emptyset \;\vee\; {}^\bullet n = \{n_p\}$

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*

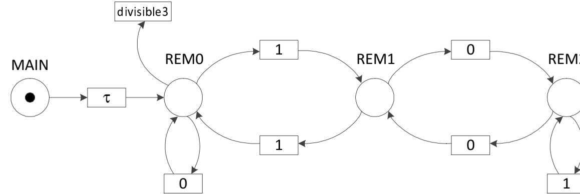


Figure 11: PN optimized associated with the specification of Example 1

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 $t \in Candidates$
    $Candidates = Candidates \backslash \{t\}$
    $NextTrans = \bigcup_{p_{next} \in t^\bullet} p_{next}^\bullet$
    if $(\exists p_{next} \in t^\bullet : \nexists \mathcal{L}_P(p_{next}) \vee \mathcal{L}_P(p_{next}) \in \{\Box, \sqcap\}) \wedge$
      $(\nexists p'_{next} \in t^\bullet : \exists t' \in {}^\bullet p'_{next} \wedge t \neq t')$
    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 $(\exists p_{next} \in t^\bullet : \nexists \mathcal{L}_P(p_{next}) \vee \mathcal{L}_P(p_{next}) \in \{\Box, \sqcap\}) \wedge$
      $(NextTrans = \{t_{next}\} \wedge \mathcal{L}_T(t_{next}) \notin \{\tau, \texttt{C1}, \texttt{C2}\} \wedge |{}^\bullet t_{next}| = 1)$
      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 $t_{next}^\bullet \neq \emptyset$
        then $F = F \cup \{(t_{prev}, p_{next}) \mid t_{prev} \in PrevTrans \wedge p_{next} \in t_{next}^\bullet\} \backslash t_{next}^\bullet$
        $P = P \backslash {}^\bullet t_{next}$, $T = T \backslash \{t_{next}\}$, $F = F \backslash \{(p, t_{next}) \in F\}$
  $P = P \backslash \{p \in P \mid (\mathcal{L}_P(p) \in \{\Box, \sqcap\} \vee \nexists \mathcal{L}_P(p)) \wedge p^\bullet = \emptyset\}$
  $T = T \backslash \{t \in T \mid \mathcal{L}_T(t) \in \{\tau, \texttt{C1}, \texttt{C2}, \|\} \wedge t^\bullet = \emptyset\}$
  $\mathcal{L}_T(t) = \tau,\ \forall t \in T : \mathcal{L}_T(t) = \| \wedge (\nexists p_{s1}, p_{s2} \in t^\bullet \mid p_{s1} \neq p_{s2})$

  return $(P, T, F, \mathcal{L}_T)$

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`*.*

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 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* $\dfrac{\Theta}{s \overset{e}{\rightarrow} s'}$ *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* $s_0 \overset{\Theta_0}{\rightsquigarrow} \ldots \overset{\Theta_n}{\rightsquigarrow} s_{n+1}$*, 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:

$$\dfrac{}{(\texttt{MAIN}, p_0, \mathcal{N}_0, \emptyset, (S_0, []), \emptyset) \overset{\tau}{\longrightarrow} (\texttt{MAIN}, p', \mathcal{N}[p_M \mapsto t_\tau \mapsto p'], (S_0, []), \emptyset)}$$

$$\dfrac{}{(\texttt{MAIN}, p', \mathcal{N}[p_M \mapsto t_\tau \mapsto p'], (S_0, []), \emptyset) \overset{\tau}{\longrightarrow} (\circlearrowleft(\texttt{MAIN}), p', \mathcal{N}[p_M \mapsto t_\tau \mapsto p'], (S_0, []), \emptyset)}$$

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 $s_0 \ldots s_{j+1}$. 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., $\nexists s_i, 0 \leq i \leq n+1 : \mathit{control}(s_i) = P \| Q$), 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 $\mathit{control}(s_{j+1})$. If the rule applied is a (Process Call - Sequential), say $\mathit{control}(s_j) = P$, then we have two possibilities:

- P was not called in the derivation $s_0 \overset{\Theta_0}{\rightsquigarrow} \ldots \overset{\Theta_{j-1}}{\rightsquigarrow} s_j$. 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 $s_0 \overset{\Theta_0}{\rightsquigarrow} \ldots \overset{\Theta_{j-1}}{\rightsquigarrow} s_j$. 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 $\mathit{control}(s_k) = P$. The first item of function LoopCheck is executed and thus $\mathit{control}(s_{k+1}) = \circlearrowleft P$. 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:*

$$size(P) = \begin{cases} 1 & \text{if } P = N \in Names \text{ or } P = STOP \\ 1 + size(P_1) & \text{if } P = a \to P_1 \\ 1 + size(P_1) + size(P_2) & \text{if } P = P_1 \sqcap P_2 \text{ or } \\ & \quad P = P_1 \Box P_2 \text{ or } P = P_1 \underset{X}{\|} P_2 \end{cases}$$

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 $\mathcal{D} = s_0 \overset{\Theta_0}{\rightsquigarrow} \ldots \overset{\Theta_n}{\rightsquigarrow} s_{n+1}$ where $s_0 = (\mathtt{MAIN}, p_0, \mathcal{N}_0, (S_0, []), \emptyset)$, 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 $\mathcal{D} = s_0 \overset{\Theta_0}{\rightsquigarrow} \ldots \overset{\Theta_n}{\rightsquigarrow} s_{n+1}$ 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).
- $P_0 \rightsquigarrow^* \mathtt{STOP}$

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:

- $P_0 \rightsquigarrow^* P || Q \rightsquigarrow^* P' || Q' \rightsquigarrow \mathtt{STOP}$, where the last rewriting step rule would correspond to the application of rule (Synchronized Parallelism 4). This case occurs, for instance, when $P' = \mathtt{STOP}$ and $Q' = \mathtt{STOP}$, 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.
- $P_0 \rightsquigarrow^* P || Q \rightsquigarrow^* \circlearrowleft(P') || Q' \rightsquigarrow^* \circlearrowleft(P') || \mathtt{STOP} \rightsquigarrow \circlearrowleft(P' || \mathtt{STOP})$, 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.

- $P_0 \rightsquigarrow^* P||Q \rightsquigarrow^* \circlearrowleft(P')||Q' \rightsquigarrow^* \circlearrowleft(P')||Q'' \rightsquigarrow P'||Q'' \rightsquigarrow^*$(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 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:

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

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 ↺ 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 $\mathcal{D} = s_0 \stackrel{\Theta_0}{\rightsquigarrow} \ldots \stackrel{\Theta_n}{\rightsquigarrow} s_{n+1}$ 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 $\mathcal{N} = (\langle \{p_0\}, \emptyset, \emptyset \rangle, M_0, \mathcal{P}, \mathcal{T}, \mathcal{L}_P, \mathcal{L}_T)$, $M_0(p_0) = 1$, $\mathcal{P} = Names \cup \{\square, \sqcap\}$, $\mathcal{T} = \Sigma^\tau \cup \{\|, \texttt{C1}, \texttt{C2}\}$.

   (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: $\mathcal{N}[p_{\mathsf{p_{prev}}} \mapsto t_\tau \mapsto p]$, 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 (Process Call - Parallel) is applied then the rule adds an internal transition t_τ and a new place p as follows: $\mathcal{N}[p_{\mathsf{p_{prev}}} \mapsto t_\tau \mapsto p]$, 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., $P = a \rightarrow Q$) 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 $p_{\Box}$ 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., $P = P_1 \underset{X}{\parallel} P_2$), 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 $t_{\parallel}$. This transition is connected to two new places (p_1 and p_2) one for each branch: $\mathcal{N}[p_{prev} \mapsto t_{\parallel} \mapsto p_1]$ and $\mathcal{N}[p_{prev} \mapsto t_{\parallel} \mapsto p_2]$, 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 $P = P_1 \underset{X}{\parallel} P_2$), 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., $P_1 \underset{X}{\parallel} P_2$), 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., $P_1 \underset{X}{\parallel} P_2$), 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 $\mathcal{N} = (\langle\{p_0\}, \emptyset, \emptyset\rangle,\ M_0, \mathcal{P}, \mathcal{T}, \mathcal{L}_P, \mathcal{L}_T),\ M_0(p_0) = 1,\ \mathcal{P} = Names \cup \{\Box, \sqcap\},\ \mathcal{T} = \Sigma^\tau \cup \{\|, \mathtt{C1}, \mathtt{C2}\}$.

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: $\mathcal{N}[p_{\mathtt{MAIN}} \mapsto t_\tau \mapsto p_1]$. τ does not belong to the set $traces(\mathtt{MAIN})$ either $L_\Sigma(\mathcal{N})$. Therefore, after this step $traces(\mathtt{MAIN}) = L_\Sigma(\mathcal{N}) = \emptyset$. 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 $traces(\mathtt{MAIN}) = L_\Sigma(\mathcal{N})$ 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 $s_n \overset{\Theta_n}{\rightsquigarrow} s_{n+1}$, $traces(\mathtt{MAIN}) = L_\Sigma(\mathcal{N})$ also holds. Let us consider the rewriting step $(P, p, \mathcal{N}, (S, S_0), \Delta) \overset{\tau}{\longrightarrow} (P', p', \mathcal{N}', (S', S_0'), \Delta')$.

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., $P = a \rightarrow Q$) 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., $P = Q \sqcap R$), 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: $\mathcal{N}[p_\sqcap \mapsto t_{C1} \mapsto p']$ or $\mathcal{N}[p_\sqcap \mapsto t_{C2} \mapsto p']$. $C1$ 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., $P_1 \underset{X}{\parallel} P_2$), 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 $t_\parallel$. 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.

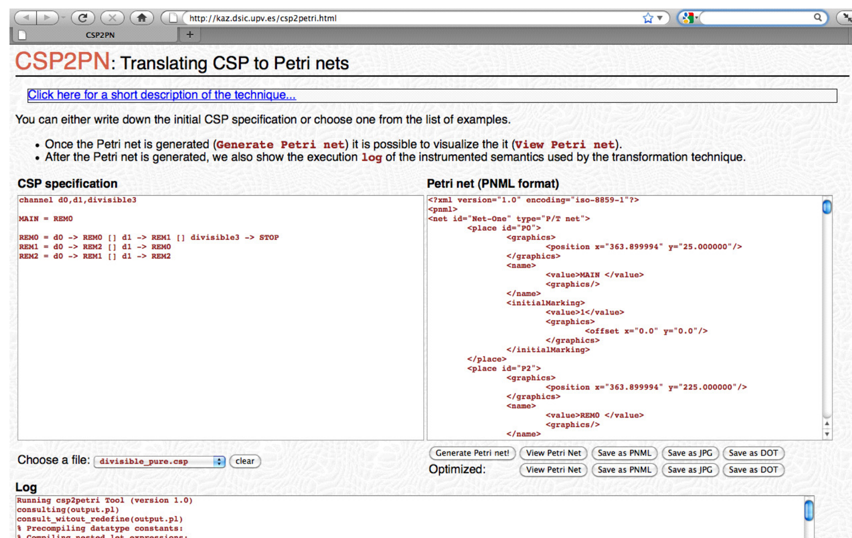


Figure 12: Screenshot of the online version of *CSP2PN*

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 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 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.