Debugging with Incomplete and Dynamically Generated Execution Trees*

David Insa and Josep Silva

Universidad Politécnica de Valencia, Camino de Vera s/n, E-46022 Valencia, Spain.
{dinsa,jsilva}@dsic.upv.es

Abstract. Declarative debugging is a powerful debugging technique that has been adapted to practically all programming languages. However, the technique suffers from important scalability problems in both time and memory. With realistic programs the huge size of the execution tree handled makes the debugging session impractical and too slow to be productive. In this work, we present a new architecture for declarative debuggers in which we adapt the technique to work with incomplete execution trees. This allows us to avoid the problem of loading the whole execution tree in main memory and solve the memory scalability problems. We also provide the technique with the ability to debug execution trees that are only partially generated. This allows the programmer to start the debugging session even before the execution tree is computed. This solves the time scalability problems. We have implemented the technique and show its practicality with several experiments conducted with real applications.

1 Introduction

Declarative debugging is a semi-automatic debugging technique that has been extended to practically all paradigms, and many techniques [9, 3, 14, 6, 5] have been defined to improve the original proposal [12]. The technique produces a dialogue between the debugger and the programmer to find the bugs. Essentially, it relies on the programmer having an *intended interpretation* of the program. In other words, some computations of the program are correct and others are wrong with respect to the programmer's intended semantics. Therefore, declarative debuggers compare the results of sub-computations with what the programmer intended. By asking the programmer questions or using a formal specification the system can identify precisely the location of a program's bug.

Traditionally, declarative debugging consists of two sequential phases: The construction of an *Execution Tree* (ET) which is an intermediate data structure that represents the execution of the program including all subcomputations; and

* This work has been partially supported by the Spanish *Ministerio de Ciencia e Innovación* under grant TIN2008-06622-C03-02, by the *Generalitat Valenciana* under grant ACOMP/2009/017, and by the *Universidad Politécnica de Valencia* (Program PAID-06-08).

the exploration of the ET with a given strategy to find the bug. A survey can be found in [15].

This technique is very powerful thanks to the ET, because it guarantees that the bug will be found whenever the programmer answers the questions of the debugger. Unfortunately, with realistic programs, the ET can be huge (indeed gigabytes) and this is the main drawback of this debugging technique, because scalability has not been solved yet: If the ET is stored in main memory, the debugger is out of memory with big ETs that do not fit. If, on the other hand, it is stored in a database, debugging becomes a slow task because some questions need to explore a big part of the ET; and also because storing the ET in the database is a time-consuming task.

Modern declarative debuggers allow the programmer to freely explore the ET with graphical user interfaces (GUI) that represent computations [4]. The scalability problem also translates to these features, because showing the whole ET (or even the part of the ET that participates in a subcomputation) is often not possible due to memory overflow reasons.

In some languages, the scalability problem is inherent to the current technology that supports the language and cannot be avoided with more accurate implementations. For instance, in Java, current declarative debuggers (e.g., JavaDD [7] and DDJ [4]) are based on the *Java Platform Debugger Architecture* (JPDA) [10] to generate the ET. This architecture uses the *Java Virtual Machine Tools Interface*, a native interface which helps to inspect the state and to control the execution of applications running in the *Java Virtual Machine* (JVM). Unfortunately, the time scalability problem described before also translates to this architecture, and hence, any debugger implemented with the JPDA will suffer the scalability problems. For instance, we conducted some experiments to measure the time needed by JPDA to produce the ET[1] of a collection of medium/large benchmarks. Results are shown in column `ET time` of Table 1. Note that, in order to generate the ET, the JVM with JPDA needs some minutes, thus the debugging session would not be able to generate the first question until this time.

In this work we propose a new implementation model that solves the three scalability problems, namely, memory, time and graphical visualization of the ET. Clearly, the ET is the bottleneck of the technique, and sometimes (e.g., in Java) it is not possible to generate it fast. Therefore, our model is based on the following question: *Is it possible to start the debugging session before having computed the whole ET?* The answer is yes.

We propose a framework in which the debugger uses the (incomplete) ET while it is being dynamically generated. Roughly speaking, two processes run in parallel. The first process generates the ET and stores it into both a database (the whole ET) and main memory (a part of the ET). The other process starts the debugging session by only using the part of the ET already generated. Moreover, we use a three-cache memories system to speedup the generation of questions

[1] These times corresponds to the execution of the program, the production of the ET and its storage in a database.

and to guarantee that the debugger is never out of memory (including the GUI components).

2 A New Architecture for Declarative Debuggers

This section presents a new architecture in which declarative debugging is not done in two sequential phases, but in two concurrent phases; that is, while the ET is being generated, the debugger is able to produce questions. This new architecture solves the scalability problems of declarative debugging. In particular, we use a database to store the whole ET, and only a part of it is loaded to main memory.

Moreover, in order to make the algorithms that traverse the ET independent of the database caching problems, we use a three-tier architecture where all the components have access to a *virtual execution tree* (VET). The VET is a data structure which is identical to the ET except that some nodes are missing (not generated yet) or incomplete (they only store a part of the method invocation) Hence, standard strategies can traverse the VET because the structure of the ET is kept.

The VET is produced while running the program. For each method invocation, a new node is added to it with the method parameters and the context before the call. The result and the context after the call are only added to the node when the method invocation finishes.

Let us explain the components of the architecture with the diagram in Figure 1. Observe that each tier contains a cache that can be seen as a view of the VET. Each cache is used for a different task:

Persistence cache. It is used to store the nodes of the VET in the database. Therefore, when the whole VET is in the database, the persistence cache is not used anymore. Basically, it specifies the maximum number of completed nodes that can be stored in the VET. This bound is called *persistence bound* and it ensures that main memory is never overflowed.

Logic cache. It defines a subset of the VET. This subset contains a limited number of nodes (in the following, *logic bound*), and these nodes are those with the highest probability of being asked, therefore, they should be retrieved from the database. This allows us to load in a single database transaction those nodes that are going to be probably asked and thus reducing the number of accesses to the database.

Presentation cache. It contains the part of the VET that is shown to the user in the GUI. The number of nodes in this cache should be limited to ensure that the GUI is not out of memory or it is too slow. The presentation cache defines a subtree inside the logic cache. Therefore, all the nodes in the presentation cache are also nodes of the logic cache. Here, the subtree is defined by selecting one root node and a depth (in the following, *presentation bound*).

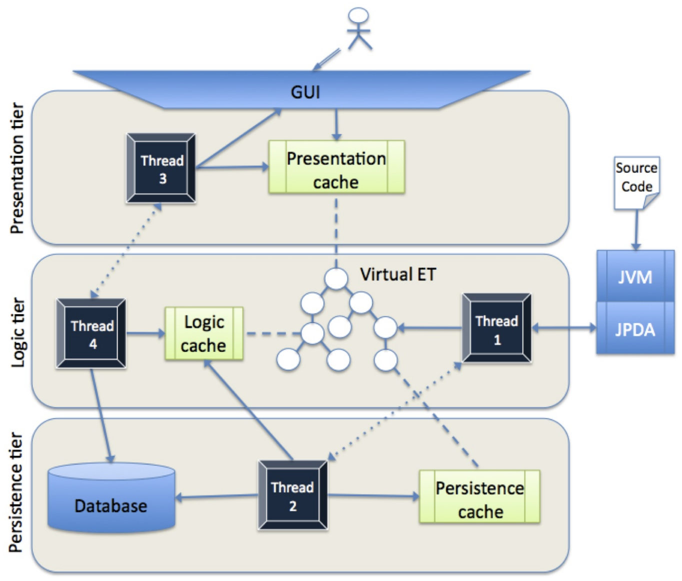
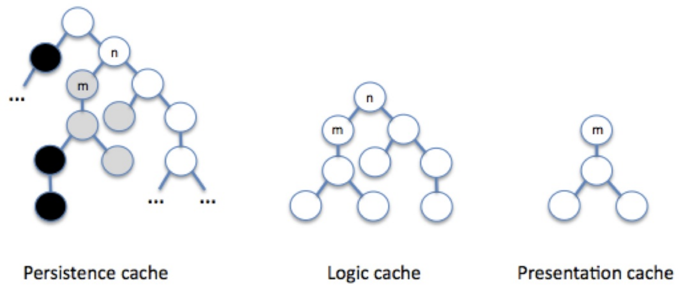


Fig. 1. Architecture of a scalable declarative debugger

The whole VET does not usually fit in main memory. Therefore, a mechanism to remove nodes from it and store them in a database is needed. When the number of complete nodes in the VET is close to the persistence bound, some of them are moved to the database, and only their identifiers remain in the VET. This allows the debugger to keep the whole ET structure in main memory and use identifiers to retrieve nodes from the database when needed.

Example 1. Consider the following trees:



The tree at the left is the VET of a debugging session where gray nodes are those already completed (their associated method invocation already finished); black nodes are completed nodes that are only stored in the database (only their identifiers are stored in the VET), and white nodes are nodes that have not been completed yet (they represent a method invocation that has not finished yet). It could be possible that some of the white nodes had a children not generated yet. Note that this VET is associated with an instant of the execution; and new nodes could be generated later. The tree in the middle is the part of the VET referenced by the logic cache, in this case it is a tree, being n the root node and a depth of four, but in general it could contain unconnected nodes. Similarly, the tree at the right is the part of the VET referenced by the presentation cache, with m the root node and a depth of three. Note that the presentation cache is a subset of the logic cache.

The behavior of the debugger is controlled by four threads that run in parallel, one for the presentation tier (thread 3), two for the logic tier (threads 1 and 4) and one for the persistence tier (thread 2). Threads 1 and 2 control the generation of the VET and its storage in the database. They collaborate via synchronizations and message passing. Threads 3 and 4 communicate with the user and generate the questions. They also collaborate and are independent of threads 1 and 2. A description of the threads and their behavior specified with pseudo-code follows:

Thread 1 (Contruction of the VET) This thread is in charge of constructing the VET. It is the only one that communicates with the JPDA and JVM. Therefore, we could easily construct a declarative debugger for another language (e.g., C++) by only replacing this thread. Basically, this thread executes the program and for every method invocation performed, it constructs a new node stored in the VET. When the number of complete nodes (given by function *completeNodes*) is close to the persistence bound, this thread sends to thread 2 the *wake up* signal. Then, thread 2 moves some nodes to the database. If the persistence bound is reached, thread 1 sleeps until enough nodes have been removed from the VET and it can continue generating new nodes.

Algorithm 1 Contruction of the VET (Thread 1)

```
Input: A source program $\mathcal{P}$
Output: A VET $\mathcal{V}$
Initialization: $\mathcal{V} = \emptyset$

repeat
(1) Run $\mathcal{P}$ with JPDA and catch event $e$
    case $e$ of
    new method invocation $I$:
(2)      create a new node $N$ with $I$
(3)      add $N$ to $\mathcal{V}$
    method invocation $I$ ended:
(4)      complete node $N$ associated with $I$
(5)      If completeNodes($\mathcal{V}$)== persistenceBound/2
(6)      then send to thread 2 the wake up signal
(7)      If completeNodes($\mathcal{V}$)== persistenceBound
(8)      then sleep
until $P$ finishes or the bug is found
```

Thread 2 (Controlling the size of the VET) This thread ensures that the VET always fits in main memory. It controls what nodes of the VET should be stored in main memory, and what nodes should be stored in the database. When the number of completed nodes in the VET is close to the persistence bound thread 1 wakes up thread 2 that removes some[2] nodes from the VET and copies them to the database. It uses the logic cache to decide what nodes to store in the database. Concretely, it tries to store in the database as many nodes as possible that are not in the logic cache, but always less than the persistence bound divided by two. When it finishes, it sends to thread 1 the *wake up* signal and sleeps.

Algorithm 2 Controlling the size of the VET (Thread 2)

```
Input: A VET $\mathcal{V}$
Output: An ET stored in a database

repeat
1) Sleep until wake up signal is received
    repeat
2)       Look into the persistence cache for the next completed node $N$ of the VET
3)       if $N$ is not found
4)       then wake up thread 1
5)              break
6)       else store $N$ in the database
7)       if $N$ is the root node then exit
```

Thread 3 (Interface communication) This thread is the only one that communicates with the user. It controls the information shown in the GUI with the presentation cache. According to the user's answers, the strategy selected, and the presentation bound, this thread selects the root node of the presentation cache. This task is done question after question according to the programmer answers, ensuring that the question asked (using function *AskQuestion*), its parent, and as many descendants as the presentation bound allows, are shown in the GUI.

Algorithm 3 Interface communication (Thread 3)

```
Input: Answers of the user
Output: A buggy node

repeat
(1) ask thread 4 to produce a question
(2) update presentation cache and GUI visualization
(3) answer = AskQuestion(question)
(4) send answer to thread 4
until a buggy node is found
```

[2] In our implementation, it removes half of the nodes. Our experiments reveal that this is a good choice because it keeps threads 1 and 2 continuously running in a producer-consumer manner.

Thread 4 (Selecting questions) This thread chooses the next question according to a given strategy using function *SelectNextQuestion* that implements standard strategies. With the node selected, the logic cache is updated and all the nodes in the logic cache are loaded from the database. This is done with function *UpdateLogicCache* that uses the node selected as the root, and the logic bound to compute the logic cache. All the nodes that belong to the new logic cache and that do not belong to the previous logic cache are loaded from the database using function *FromDatabaseToET*.

Algorithm 4 Selecting questions (Thread 4)

```
Input: A strategy $\mathcal{S}$ and a VET $\mathcal{V}$
Output: A buggy node

repeat
(1) question = SelectNextQuestion($\mathcal{V}$,$\mathcal{S}$)
(2) missingNodes = UpdateLogicCache()
(3) If (question $\notin \mathcal{V}$) then $\mathcal{V}$ = FromDatabaseToET($\mathcal{V}$,missingNodes)
(4) send question to thread 3
(5) get answer from thread 3
until a buggy node is found
```

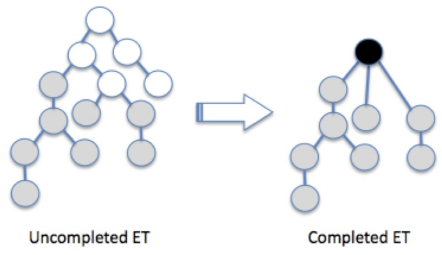
2.1 Redefining the Strategies for Declarative Debugging

In Algorithm 4, a strategy is used to generate the sequence of questions by selecting nodes in the VET. Nevertheless, all declarative debugging strategies in the literature have been defined for ETs and not for VETs where incomplete nodes can exist. All of them assume that the information of all ET nodes is available. Clearly, this is not true in our context and thus, the strategies would fail. For instance, the first node asked by the strategy top-down and its variants is always the root node of the ET. However, this node is the last node completed by Algorithm 3. Hence, these strategies could not even start until the whole ET is completed, and this is exactly the problem that we want to avoid.

Therefore, in this section we propose a redefinition of the strategies for declarative debugging so that they can work with VETs.

A first solution could be to define a transformation from a VET with incomplete nodes to a VET where all nodes are completed. This can be done by inserting a new root node with the equation $1 = 0$. Then, the children of this node would be all the completed nodes whose parent is incomplete. In this way, (i) all nodes of the produced ET would be completed and could be asked; (ii) the parent-child relation is kept in all the subtrees of the ET; and (iii) it is guaranteed that at least one bug (the root node) exists. If the debugging session finishes with the root node as buggy, it means that the node with the "real" bug (if any) has not been completed yet.

Example 2. Consider the following VETs:



In the VET at the left, gray nodes are completed, and white nodes are incomplete. This VET can be transformed into the VET at the right where all nodes are completed. The new artificial root is the black node which ensures that at least one buggy node exists.

From an implementation point of view, this transformation is inefficient and costly because the VET is being generated continuously by thread 1, and hence, this transformation should be done repeatedly question after question. In contrast, a more efficient solution is to redefine the strategies so that they ignore incomplete nodes. For instance, top-down [1] only asks for the descendants of a node that are completed and that do not have a completed ancestor. Similarly, Binks top-down [2] would ask first for the completed descendant that in turn contains more completed descendants. D&Q [12] would ask for the node that divides the VET in two subtrees with the same number of completed nodes, and so on. We refer the interested reader to the source code of our implementation that is publicly available and where all strategies have been reimplemented for VETs.

Even though the architecture presented has been discussed in the context of Java, it can work for other languages with very few changes. Observe that the part of an algorithmic debugger that is language-dependent is the front-end, and our technique relies on the back-end. Once the VET is generated, the back-end can handle the VET mostly independent of the language. In particular, the strategies can traverse the VET being unaware of the meaning of the questions.

3 Implementation

We have implemented the technique presented in this paper and integrated it into DDJ 2.4. The implementation has been tested with a collection of real applications (e.g., an interpreter, a parser, a debugger, etc).

Table 1 summarizes the results of the experiments performed. These experiments have been done in an Intel Core2 Quad 2.67 GHz with 2GB RAM.

The first column contains the names of the benchmarks. For each benchmark, the second and third columns give an idea of the size of the execution. Fourth and fifth columns are time measures. Finally, sixth and seventh columns show memory bounds. Concretely, column `variables number` shows the number of variables participating (possibly repeated) in the execution considered. Column `ET size` shows the size in Mb of the ET when it is completed, this measure

Benchmark	var. num.	ET size	ET time	node time	cache lim.	ET depth
argparser	8.812	2 Mb	22 s.	407 ms.	7/7	7
cglib	216.931	200 Mb	230 s.	719 ms.	11/14	18
kxml2	194.879	85 Mb	1318 s.	1844 ms.	6/6	9
javassist	650.314	459 Mb	556 s.	844 ms.	7/7	16
jtstcase	1.859.043	893 Mb	1913 s.	1531 ms.	17/26	57
HTMLcleaner	3.575.513	2909 Mb	4828 s.	609 ms.	4/4	17

Table 1. Benchmark results

has been taken from the size of the ET in the database (of course, it includes compaction). Column `ET time` is the time needed to finish the ET. Column `node time` is the time needed to complete the first node of the ET. Column `cache limit` shows the presentation bound and the depth of the logic cache of these benchmarks. After these bounds, the computer was out of memory. Finally, column `ET depth` shows the depth of the ET after it was constructed.

Observe that a standard declarative debugger is hardly scalable to these real programs. With the standard technique, even if the ET fits in main memory or we use a database, the programmer must wait for a long time until the ET is completed and the first question can be asked. In the worst case, this time is more than one hour. Contrarily, with the new technique, the debugger can start to ask questions before the ET is completed. Note that the time needed to complete the first node is always less than two seconds. Therefore, the debugging session can start almost instantaneously.

The last two columns of the table give an idea of how big is the ET shown in the GUI before it is out of memory. In general, five levels of depth is enough to see the question asked and the part of the computation closely related to this question. In the experiments only HTMLcleaner was out of memory when showing five levels of the ET in the GUI.

All the information related to the experiments, the source code of the benchmarks, the bugs, the source code of the tool and other material can be found at http://www.dsic.upv.es/~jsilva/DDJ

4 Conclusions

Declarative debugging is a powerful debugging technique that has been adapted to practically all programming languages. The main problem of the technique is its low level of scalability both in time and memory. With realistic programs the huge size of the internal data structures handled makes the debugging session impractical and too slow to be productive.

In this work, we propose the use of VETs as a suitable solution to these problems. This data structure has two important advantages: It is prepared to be partially stored in main memory, and completely stored in secondary memory. This ensures that it will always fit in main memory and thus solves the memory scalability problem. In addition, it can be used during a debugging session before it is completed. For this, we have implemented a version of standard declarative debugging strategies able to work with VETs. This solves the time scalability problem as demonstrated by our experiments.

In our implementation, the programmer can control how much memory is used by the GUI components, and by the strategies thanks to the use of three cache memories. The most important result is that experiments confirm that, even with large programs and long running computations, a debugging session can start to ask questions after only few seconds.

References

1. E. Av-Ron. *Top-Down Diagnosis of Prolog Programs.* PhD thesis, Weizmanm Institute, 1984.
2. D. Binks. *Declarative Debugging in Gödel.* PhD thesis, University of Bristol, 1995.
3. R. Caballero. A Declarative Debugger of Incorrect Answers for Constraint Functional-Logic Programs. In *Proc. of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming (WCFLP'05)*, pages 8–13, New York, USA, 2005. ACM Press.
4. R. Caballero. Algorithmic Debugging of Java Programs. In *Proc. of the 2006 Workshop on Functional Logic Programming (WFLP'06)*, pages 63–76. Electronic Notes in Theoretical Computer Science, 2006.
5. R. Caballero, N. Martí-Oliet, A. Riesco, and A. Verdejo. A declarative debugger for maude functional modules. *Electronic Notes Theoretical Computer Science*, 238(3):63–81, 2009.
6. T. Davie and O. Chitil. Hat-delta: One Right Does Make a Wrong. In *Seventh Symposium on Trends in Functional Programming, TFP 06*, April 2006.
7. H. Girgis and B. Jayaraman. JavaDD: a Declarative Debugger for Java. Technical Report 2006-07, University at Buffalo, March 2006.
8. G. Kokai, J. Nilson, and C. Niss. GIDTS: A Graphical Programming Environment for Prolog. In *Workshop on Program Analysis For Software Tools and Engineering (PASTE'99)*, pages 95–104. ACM Press, 1999.
9. I. MacLarty. *Practical Declarative Debugging of Mercury Programs.* PhD thesis, Department of Computer Science and Software Engineering, The University of Melbourne, 2005.
10. Sun Microsystems. Java Platform Debugger Architecture - JPDA. Available from URL: `http://java.sun.com/javase/technologies/core/toolsapis/jpda/`.
11. H. Nilsson and P. Fritzson. Algorithmic Debugging for Lazy Functional Languages. *Journal of Functional Programming*, 4(3):337–370, 1994.
12. E.Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, 1982.
13. J. Silva. An Empirical Evaluation of Algorithmic Debugging Strategies. Technical Report DSIC-II/10/09, UPV, 2009. Available from URL: `http://www.dsic.upv.es/~jsilva/research.htm#techs`.
14. J. Silva. Algorithmic debugging strategies. In *Proc. of International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2006)*, pages 134–140, 2006.
15. J. Silva. A Comparative Study of Algorithmic Debugging Strategies. In *Proc. of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'06)*, pages 143–159. Springer LNCS 4407, 2007.