Document downloaded from:

http://hdl.handle.net/10251/35503

This paper must be cited as:

Insa Cabrera, D.; Silva Galiana, JF. (2011). Scaling up algorithmic debugging with virtual execution trees. En Logic-Based Program Synthesis and Transformation. Springer Verlag (Germany). 6564:149-163. doi:10.1007/978-3-642-20551-4_10.

The final publication is available at

http://link.springer.com/chapter/10.1007/978-3-642-20551-4_10

Copyright Springer Verlag (Germany)

Scaling up Algorithmic Debugging with Virtual 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].

This technique is very powerful thanks to the use of an internal data structure called *Execution Tree* (ET) that represents all computations of a given program. Unfortunately, with realistic programs, the ET can be huge (indeed gigabytes) and this is the main drawback of this debugging technique, because the lack of scalability has not been solved yet: If ET's are stored in main memory, the debugger is out of memory with big ET's that do not fit. If, on the other hand, they are 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.

* This work has been partially supported by the Spanish *Ministerio de Ciencia e Innovación* under grant TIN2008-06622-C03-02, by the *Generalitat Valenciana* under grant ACOMP/2010/042, and by the *Universidad Politécnica de Valencia* (Program PAID-06-08).

Modern declarative debuggers allow the programmer to freely explore the ET with graphical user interfaces (GUI) [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 (e.g., an interpreter, a parser, a debugger, etc). Results are shown in column `ET time` of Table 1.

`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

Note that, in order to generate the ET, the JVM with JPDA needs some minutes, thus the debugging session would not be able to start 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. Because it is not always possible (e.g., in Java) to generate the ET fast, the process of generating the ET can cause a bottleneck in the declarative debugging. 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,

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

we use a three-cache memories system to speed up the debugging session and to guarantee that the debugger is never out of memory (including the GUI components).

The rest of the paper has been structured as follows: In Section 2 we recall the declarative debugging technique with an example. In Section 3 we introduce a new implementation architecture for declarative debuggers and discuss how it solves the three scalability problems. In Section 4 we give some details about the implementation and show the results obtained with a collection of benchmarks. Finally, Section 5 concludes.

2 Declarative Debugging

Traditionally, declarative debugging consists of two sequential phases: (1) The construction of the ET that represents the execution of the program including all subcomputations; and (2) the exploration of the ET with a given search strategy to find the bug.

In the second phase, 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.

The ET contains nodes that represent subcomputations of the program. Therefore, the information of the ET's nodes is different in each paradigm (e.g., functions, methods, procedures, etc.), but the construction of the ET is very similar in all of them [13]. In the object oriented paradigm, the ET is constructed as follows: Each node of the ET is associated to a method invocation, and it contains all the information needed to decide whether the method invocation produced a correct result. This information includes the call to the method with its parameters and the result, and the values of all the attributes that are in the scope of this method, before and after the invocation. This information allows the programmer to know whether all the effects of the method invocation correspond to the intended semantics. The root node of the ET can be any method invocation, typically, the *main* method of the program. For each node n with associated method m, and for each method invocation m' done by m, a new node associated to m' is recursively added to the ET as the child of n.

Example 1. Consider the Java program in Figure 1. This program takes an array of integers and adds to each element its factorial. The ET of this program taking as root the call to $s.sum$ made by $main$ is shown in Figure 2. Note that, in the object-oriented paradigm, a method invocation could produce a result, a change in the state, or both.

```
public class SumFactorial {
  int[] v = {1,3,2};

  public static void main() {
    SumFactorial s = new SumFactorial();
    s.sum();
  }

  public void sum() {
    int y;
    for(int i = 0; i < v.length; i++){
      y=v[i];
      v[i]=factorial(v[i])+y;}
  }

  public static int factorial(int x) {
    if (x==0 || x==1) return 1;
    else return (x-1) * factorial(x-1);
  }
}
```

Fig. 1. Example program with a bug

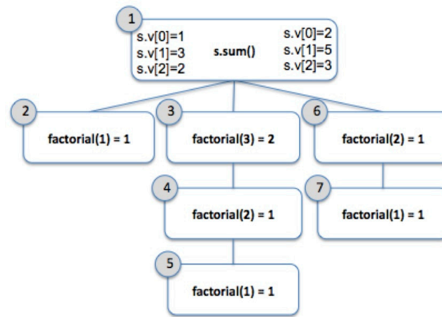


Fig. 2. ET associated with the program in Figure 1

For the purpose of this work, we can consider ET's as labeled trees where labels are strings that represent method invocations with their results and their effects on the context. Formally,

Definition 1 (Context). *Let $\mathcal{P}$ be a program, and m a method in $\mathcal{P}$. Then, the* context *of m is $\{(a, v) \mid a$ is an attribute in the scope of m and v is the value of $a\}$.*

Roughly, the context of a method is composed of all the variables of the program that can be affected by the execution of this method. These variables can be other objects that in turn contain other variables. In a realistic program, each node contains several data structures that could change during the invocation. All this information (before and after the invocation) should be stored together with the call to the method so that the programmer can decide whether it is correct.

Definition 2 (Method Invocation). *Let $\mathcal{P}$ be a program and $\mathcal{E}$ an execution of $\mathcal{P}$. Then, each* method invocation *of $\mathcal{E}$ is represented with a triple $I = (b, m, a)$ where m is a string representing the call to the method with its parameters and the returned value, b is the context of the method in m before its execution and a is the context of the method in m after its execution.*

Method invocations are the questions asked by the debugger. For instance, a method invocation $I = (b, m, a)$ corresponds to the question: *Is the final context a correct, if we execute method m with context b?* The answer of the user can be YES (to denote that this computation is correct) or NO (to denote that the computation is wrong). For the sake of concreteness, we ignore other possible answers such as "I don't know" or "I trust this function". They are accepted in our implementation, but we refer the interested reader to [15] for theoretical implications of their use.

Definition 3 (Execution Tree). *Given a program $\mathcal{P}$ with a set of method definitions $M = \{m_i \mid 1 \leq i \leq n\}$, and a call c to $m \in M$, the* execution tree *(ET) of $\mathcal{P}$ with respect to c is a tree $t = (V, E)$ where the label of a node $v \in V$ is denoted with $l(v)$. $\forall v \in V, l(v) = i$ where i is a method invocation. And*

- *The root of the ET is the method invocation associated with c.*
- *For each node associated with a call c' to $m_j, 1 \leq j \leq n$, there is a child node associated with a call c'' to $m_k, 1 \leq k \leq n$, if and only if*
  1. *during the execution of c', c'' is invoked, and*
  2. *the call c'' is done from the definition of m_j.*

Once the ET is built, in the second phase, the debugger uses a search strategy to traverse the ET to find for the buggy node (a buggy node is associated with the buggy source code of the program). During the search, the debugger asks the programmer to answer the question associated with some nodes. At the beginning, the *suspicious area* which contains those nodes that can be buggy is the whole ET; but, after every answer, some nodes of the ET leave the suspicious area. When all the children of a node with a wrong equation (if any) are correct, the node becomes buggy and the debugger locates the bug in the method of the program associated with this node [11]. If a bug symptom is detected then declarative debugging is complete [12]; hence, if all the questions are answered, the bug will eventually be found. An explanation of the behavior of current algorithmic debugging strategies can be found in [15].

Due to the fact that questions are asked in a logical order, *top-down search* [1] is the strategy that has been traditionally used by default by different debugging tools and methods. It basically consists of a top-down, left-to-right traversal of the ET. When a node is answered NO, one of its children is asked; if it is answered YES, one of its siblings is asked, thus, the node asked is always a child or a sibling of the previous question node. Therefore, the idea is to follow the path of wrong equations from the root of the tree to the buggy node.

However, selecting always the left-most child does not take into account the size of the subtrees that can be explored. Binks proposed in [2] a variant of top-down search in order to consider this information when selecting a child. This

variant is called *heaviest first* because it always selects the child with a bigger subtree. The objective is to avoid selecting small subtrees which have a lower probability of containing a bug.

The generalization of this idea to the whole tree is implemented by the strategy *divide & query* (D&Q) that always asks for the node that divides the ET in two parts as similar as possible. Therefore, D&Q has an cost in the number of questions $O(n * log\ n)$ being n the number of nodes of the ET.

Example 2. Two declarative debugging sessions for the ET in Figure 1 are shown below. The left session corresponds to the strategy top-down; the right session corresponds to D&Q. `YES` and `NO` answers are provided by the programmer:

```
Starting Debugging Session...
(1)  s.v=[1,3,2] s.sum() s.v=[2,5,3]? NO
(2)  factorial(1) = 1? YES
(3)  factorial(3) = 2? NO
(4)  factorial(2) = 1? NO

Bug found in method:
SumFactorial.factorial()
```

```
Starting Debugging Session...
(3)  factorial(3) = 2? NO
(5)  factorial(1) = 1? YES
(4)  factorial(2) = 1? NO

Bug found in method:
SumFactorial.factorial()
```

The debugger points out the part of the code which contains the bug. In this case, `return (x-1) * factorial(x-1)` should be `return x * factorial(x-1)`. Note that, to debug the program, the programmer only has to answer questions. It is not even necessary to see the code.

3 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 search 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 3. 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:

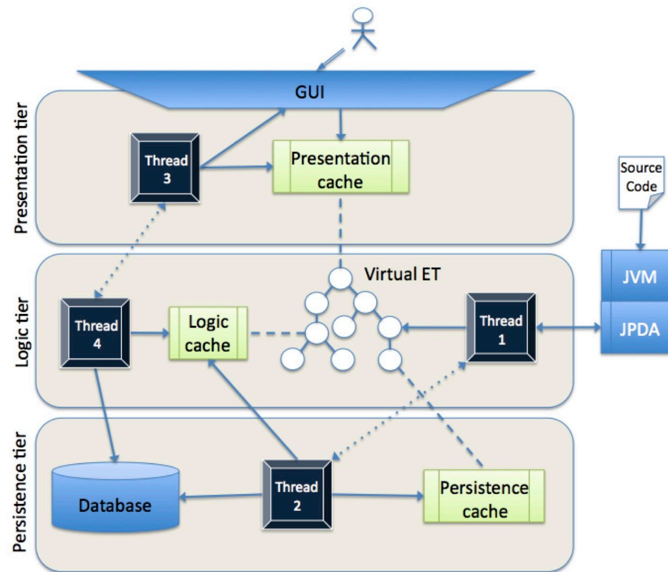


Fig. 3. Architecture of a scalable declarative debugger

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.

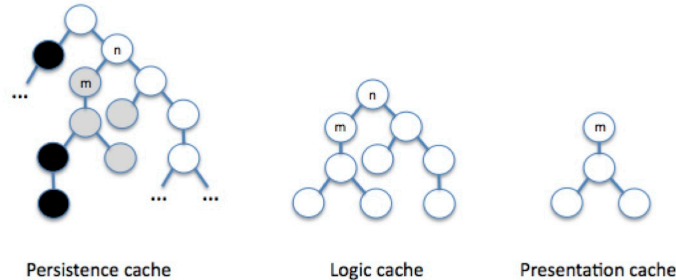
Logic cache. It defines a subset of the VET. This subset contains a limited number of nodes (in the following, *logic bound*), and these nodes are those with the highest probability of being asked, therefore, they should be retrieved from the database. This allows us to load in a single database transaction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

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 3. 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 children not generated yet. Note that this VET is associated with an instant of the execution; and new nodes could be generated or completed 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 (i.e., shown in the GUI), 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 (Construction 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 Construction of the VET (Thread 1)

Input: A source program $\mathcal{P}$, and the persistence bound $persistenceBound$
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 $\mathcal{P}$ 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
until the whole VET is stored in the database or until the bug 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 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 select a node
(2) update presentation cache and GUI visualization
(3) answer = AskQuestion(node)
(4) send answer to thread 4
until a buggy node is found

Thread 4 (Selecting questions) This thread chooses the next node according to a given strategy using function *SelectNextNode* that implements any standard search strategy. If the node selected is not loaded in the logic cache (*not(InLogicCache(node))*) the logic cache is updated. This is done with function *UpdateLogicCache* that uses the node selected and the logic bound to compute the new logic cache (i.e, those nodes of the VET that should be loaded from the database). 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 $\mathcal{S}$, a VET $\mathcal{V}$ and the logic bound $logicBound$
Output: A buggy node

repeat
(1) node = SelectNextNode($\mathcal{V}$,$\mathcal{S}$)
(2) **If** not(InLogicCache(node)) **then** missingNodes = UpdateLogicCache(node,logicBound)
(3) $\mathcal{V}$ = FromDatabaseToET($\mathcal{V}$,missingNodes)
(4) send node to thread 3
(5) get answer from thread 3 and change the state of the nodes affected
until a buggy node is found

An interesting task of this thread is carried out by function $UpdateLogicCache$ that determines what nodes should remain in the VET when a new node is selected by the strategy used. This task is not critical for the performance of the debugger. The bottleneck is the construction of the VET; contrarily, the ex-

ploration of the VET is a quick task that can be done step by step after each answer of the oracle. However, the caching policy should be efficient to optimize the resources of the debugger, and to minimize the number of accesses to the database. In particular, those nodes that are already cached in the VET and that could be needed in future questions should remain in the VET. Contrarily, those nodes that are definitely discarded by the answers of the user should be replaced by other nodes that can be useful.

Although each search strategy uses a different caching policy, all of them share four invariants: (i) When a node is marked as wrong, then all the nodes that are not descendants of this node are useless, and they can be unloaded from the VET. (ii) When a node is marked as right, then all the descendants of this node are useless, and they can be unloaded from the VET. (iii) When a node is marked as wrong, then all the descendants of this node that have not been already discarded, could be needed in the future, and they remain in the VET. (iv) When a node is marked as right, then all the nodes that are not descendants of this node (and that have not been already discarded) could be needed in the future, and they remain in the VET.

As an example, with the strategy Top-Down, initially, we load in the VET the root node and those nodes that are close to the root. Every time a node is marked as correct, all the nodes in the subtree rooted at this node are marked as useless and they can be unloaded from the VET when it is needed. All the nodes that have been already answered, except the last node marked as wrong can be also unloaded. When the strategy is going to select a node that is not loaded in the VET, then all the useless nodes are unloaded from the VET, and those nodes that are descendant of the selected node and that are closer to it are loaded. In this way, the number of accesses to the database are minimized, because every load of nodes to the VET provides enough nodes to answer several questions before a new load is needed.

3.1 Redefining the Strategies for Declarative Debugging

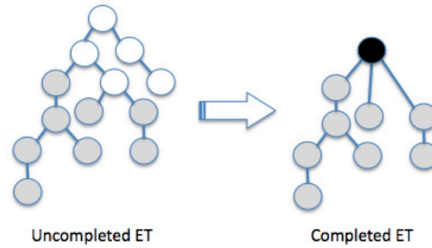
In Algorithm 4, a strategy is used to generate the sequence of questions by selecting nodes in the VET. Nevertheless, all declarative debugging strategies in the literature have been defined for ET's and not for VET's where incomplete nodes can exist. All of them assume that the information of all ET 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 1. 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 VET's.

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 4. Consider the following VET's:



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 completed node that divides the VET in two subtrees with the same number of 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 VET's.

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. We provide a discussion of the changes needed to adapt the architecture to other languages in the next section.

4 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. 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. It is the sum of all variables in all the nodes of the ET. Column `ET size` shows the total size in Mb of the ET when it is completed, this measure 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 completely generate 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/\~~\hspace{-0.7em}jsilva/DDJ`

4.1 Adapting the Architecture for other Languages

Even though the technique presented is based on the Java language, it is mostly independent of the language that is being debugged. Note that all algorithmic debuggers are based on the ET and the main difference between debuggers for different languages is the information stored in each node (e.g., functions in the functional paradigm, methods in the object-oriented paradigm, functions and procedures in the imperative paradigm, predicates in the logic paradigm, etc.). But once the ET is constructed, the strategies that traverse the ET are independent of the language the ET represents.

Therefore, because our technique is based on the structure of the ET, but not on the information of its nodes, all the components of the architecture presented in Figure 3 could be used for other languages with small changes. In the case of object-oriented languages such as C++ or C#, the only component that needs to

be changed is thread 1. In other languages, the changes needed would involve the redefinition of the components to change the kind of information (e.g., clauses, predicates, procedures, etc.) stored (in the database and in the virtual ET) and shown (in the GUI). However, the behavior of all the components and of all the threads would be exactly the same. The only part of the architecture that should be completely changed is the part in charge of the construction of the ET. JPDA is exclusive for Java, hence, thread 1 should be updated to communicate with other component similar to the JPDA but for other language. All the interrelations between the components would remain unchanged. Finally, another small modification would be done in the GUI. Our implementation shows to the user objects, methods and attributes. Showing procedures, global variables or functions would be very similar, but small changes would still be needed.

5 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 VET's 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 it 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 VET's. 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 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.

A debugging session that is performed while the ET is being constructed could ask more questions than a classical debugging session. However, the time used in the classical debugging session is greater than or equal to the time needed in the new architecture. It should be clear that the goal of our technique is not to minimize the number of questions. The goal is to minimize the debugging time. Our technique tries to use the time used to build the ET to also explore the ET in order to search the bug from the first second. In the best case, the bug will be found even before the VET is constructed. In the worst case, when the VET is finished, the bug will remain hidden, but several questions will be already answered, and thus the search space will be smaller. Therefore, even in the worst case, our technique improves the time needed to find the bug.

6 Acknowledgements

We thank the anonymous referees of LOPSTR 2010 for providing us with constructive comments and suggestions which contributed to improving the quality of the work.

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, C. Hermanns, and H. Kuchen. 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. I. MacLarty. *Practical Declarative Debugging of Mercury Programs.* PhD thesis, Department of Computer Science and Software Engineering, The University of Melbourne, 2005.
9. Sun Microsystems. Java Platform Debugger Architecture - JPDA, 2010. Available from URL: `http://java.sun.com/javase/technologies/core/toolsapis/jpda/`.
10. H. Nilsson and P. Fritzson. Algorithmic Debugging for Lazy Functional Languages. *Journal of Functional Programming*, 4(3):337–370, 1994.
11. E.Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, 1982.
12. J. Silva. An Empirical Evaluation of Algorithmic Debugging Strategies. Technical Report DSIC-II/10/09, UPV, 1998. Available from URL: `http://www.dsic.upv.es/~jsilva/research.htm#techs`.
13. J. Silva. Algorithmic debugging strategies. In *Proc. of International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2006)*, pages 134–140, 2006.
14. 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.