Embee Performance Tests
Technical Report 2003-465

Michelle L. Crane and Juergen Dingel
School of Computing
Queen's University
Kingston, Ontario, Canada
crane@cs.queensu.ca
dingel@cs.queensu.ca

Abstract

We have designed and implemented a prototype tool, called Embee, which makes use of the Alloy Analyzer to automatically check the conformance of Java executions against Alloy specifications. Running time tests were conducted as part of the performance analysis of this tool. This report describes these tests and their results. Some of the results were unexpected, leading to questions about how the Alloy Analyzer functions.

1 Motivation

In the course of researching "Runtime Conformance Checking of Objects Using Alloy" [1, 2], we have designed and implemented a prototype tool, which can be used to check the conformance of a Java program's execution against an Alloy [4, 6] specification. This tool, called Embee, makes use of the Alloy Analyzer [3, 7] to automatically check the conformance of a target implementation against structural constraints specified in Alloy. A series of running time tests were conducted as part of the performance analysis of our prototype.

The purpose of this report is to document the motivation for testing and the test series used. In addition, we will interpret the results. Finally, some of these results were unexpected; we will discuss these and outline future work to study the discrepancies.

2 Background

Execution of Embee occurs in three phases. The first phase focusses on creating a static mapping between the naming scheme used in the Alloy specification and that used in the Java implementation. The second phase makes use of the JPDA [8] to execute the target implementation and halt it at user-specified breakpoints. The objects that exist at the top of the target's runtime stack are examined and information about these objects, and the relationships between them, is output to a series of textual dump files.

In the third phase, Embee creates a set of empty dynamic mappings representing all of the objects and relations that could exist in a given finite scope. Then, using the information from an individual dump file, Embee populates these dynamic mappings and extracts a Boolean array representing the Alloy atoms and relations that exist at that particular breakpoint. This array is passed, along with the original specification, to the Alloy Analyzer. The Analyzer determines whether or not the array represents a satisfying instance of the specification. This third phase actually mimics the functionality of the Analyzer's 'Edit Instance' command, with the exception that Embee edits the instance automatically.

3 Objectives

After Embee was implemented, a series of performance tests were executed in order to examine the running time of each phase.

3.1 Primary Objective

The primary objective of these tests was to explore the running times of each phase of Embee, with a particular emphasis on Phase 3, i.e., conformance checking. Our goal was to demonstrate that the computations performed by Embee to create the Boolean array in Phase 3 required less running time than the Analyzer's computations to check that array against the specification.

3.2 Secondary Objectives

We also had several secondary objectives relating to how the running time of the conformance check is affected by the scope of analysis and the complexity of the specification. The following objectives are presented in reverse order of importance:

1. Evaluate the effect of the scope of analysis on the running time of Phase 3. The scope of analysis for any particular breakpoint is a finite number, equal to the maximum number of objects of any one type found at that breakpoint. We expect that the higher the scope of analysis, the longer the running time of the conformance check.
2. Evaluate the effect of the number of explicit facts on the running time of Phase 3. The specifications in our test families are distinguished by the number of explicit facts in the specification; Section 4.1 discusses this concept. We expect that the conformance check running time will increase with the number of explicit facts within a particular test family.
3. Determine whether or not a certain, simple estimate of the complexity of a specification is a valid measure of the running time required to complete Phase 3. The complexity of the Analyzer's portion of this check has been described as linear [9] in the size of the Boolean formula representing the specification (in a certain scope). We would therefore expect that a specification with a longer formula would take longer to check than one with a shorter formula. Unfortunately, we do not have the means to determine the exact length of these formulas; therefore, we attempt to estimate the length of the formulas. We decided to use the number of operators in the formula as it is passed to the Analyzer, as discussed in Section 4.1.1. Our last secondary objective is to determine whether or not this simple estimate is useful for determining the running time of our test cases.

4 Testing

4.1 Methodology

Our tests focus on three factors, or dimensions, affecting the running time of the conformance checking phase:

1. First, we performed the tests on three families of specification, each one of which describes a simple data structure, such as a list, graph or tree. The specifications for these three structures are listed in Appendix A. Each of these three family groups contains different numbers of signatures and relations, with different arities of relations. In addition, within each family group, there were several specifications, identical except for the number of explicit facts.
2. The second dimension is the number of explicit facts in a particular specification. Explicit facts are those which appear in the second set of braces

of a signature or as separate (named) facts. These explicit facts were used to create sets of specifications with different complexities, e.g., a list specification with two explicit facts would be considered more complex than the same specification with no explicit facts.

3. The final dimension is the number of objects at each breakpoint. We created implementations of each of the three data structures; for each data structure, we create a large number of objects, e.g., a list with many nodes, etc. A breakpoint is encountered at the end of each add method; the conformance of the implementation at each breakpoint was then checked. The number of node objects existing at a particular breakpoint becomes the scope of the analysis. Finally, the target implementations were designed to conform to the specification; this forces the Analyzer to evaluate all of the objects and their relationships, resulting in longer (worst-case) running times.

When executing Embee, an `OutOfMemory` error occasionally occurs, for instance, when the specification gets overly complex, or when many objects are being checked for conformance. This is an error caused by the fact that the Java Virtual Machine (JVM) reserves a certain amount of memory for itself; the default is 64MB. It is quite simple for a conformance check to require more than this amount of memory, for example, with a large number of dynamic mappings and a complicated Boolean formula. In order to provide an equal baseline for all tests, we ran the conformance check with additional memory for the JVM—256MB to be exact. All tests were performed on a Pentium III, 650MHz, with a total of 512MB of RAM.

4.1.1 Estimating the Length of the Boolean Formula

The Boolean formula represents the original Alloy specification and is affected by both the complexity of the specification itself and the scope of the analysis. It is possible to retrieve a textual representation of the formula from the Alloy Analyzer. We have created a small utility, which extracts a count of the number of Boolean operators ("and", "or", and "not") from the formula. We estimate the length of the formula as the total count of these operators.

For example, Table 1 contains the estimates for our three test families, from *scope* = 1 through *scope* = 4. The size of the formula quickly becomes very large (and our simple utility is extremely inefficient and thus quickly runs out of memory); therefore, we did not compute our estimates past scope 4.

Table 1: Estimate of Boolean formula size, determined by number of Boolean operators (“and”, “or”, “not”)

Example 1 - List			
scope	0 Facts	1 Fact	2 Facts
1	23	34	43
2	197	657	729
3	671	13,799	15,200
4	1,731	91,435	96,771

Example 2 - Graph				
scope	Facts	1 Fact	2 Facts	3 Facts
1	—	—	—	—
2	185	1,005	1,783	2,181
3	674	66,722	118,250	142,328
4	1,787	635,811	1,153,063	1,319,611

Example 3 - Tree					
scope	0 Facts	1 Fact	2 Facts	3 Facts	4 Facts
1	39	78	93	103	104
2	367	1,601	2,487	2,629	2,715
3	1,283	38,528	73,472	76,196	76,568
4	3,359	234,595	456,459	466,979	468,087

For the most part, we can see that adding explicit facts to a particular specification increases the number of operators in the related Boolean formula. In some cases, the increase is not great, but the additional explicit fact never decreases the length of the formula. The size of the formula is largest for the *Graph* specifications and smallest for the *List* specifications, so we would expect the *Graph* conformance checks to take the most time and the *List* conformance checks to take the least time.

4.2 Test Series

Table 2 summarizes the series of tests performed to determine the running time of the conformance checking phase. Our intention was to test each series up to a scope of 40. However, in some cases, we were forced to reduce the size of the scope, such as when an `OutOfMemory` occurred even with 256MB, or when the test ran too long (we imposed a limit of one hour for these tests). Finally, some series were aborted when the conformance check became unstable; this issue is discussed in Section 5.1.2.

Table 2: Test series for evaluating the running time of conformance checking

Test Family	Data Structure	S^1	R^2	arity$(r_i)^3$	Explicit Facts	*scope*	Number of Tests
List	singly linked list	2	2	2, 2	0	1–40	40
					1	1–32	32
					2	1–31	31
Graph	directed acyclic graph	2	2	2, 3	0	2–40	39
					1	2–40	39
					2	2–34	33
					3	2–34	23
Tree	binary tree	3	4	2, 2, 2, 2	0	1–40	40
					1	1–40	40
					2	1–32	32
					3	1–32	32
					4	1–32	32
Total Number of Tests (Conformance Checks)							413

[1] S is the number of signatures which are used in the specification.
[2] R is the number of relations (or fields) in the specification.
[3] Each relation has an arity, e.g., binary, ternary, etc. The arity(r_i) column contains the arity of all of the relations in the specification.

5 Results

5.1 General Results

Table 3 contains the running times of each phase for three of our test cases. These times give a general feel of how the fast Embee works in its entirety. In each case, a data structure was created with 20 nodes. As can be seen from the table, the conformance checking phase (Phase 3) takes the longest to complete. Interestingly enough however, even this phase does not take long when the scope is kept to a reasonable size, i.e., $scope \leq 16$. Checking the first 16 breakpoints takes significantly less time than checking the last 4 breakpoints, regardless of the complexity of the specification.

Table 3: Running times for each phase and total running time of Embee

Test Case			Running Time (m:ss)				
Object Model	Scope	Number of Breakpoints	Phase 1	Phase 2	Phase 3		Total
					First 16	Last 4	
List	20	20	0:07	0:32	0:12	6:39	07:30
Graph	20	19	0:07	1:27	0:35	44:10	46:19
Tree	20	20	0:04	1:20	0:21	6:04	07:49

5.1.1 Phase 3

Figure 1 shows the running times for the conformance checking phase for all of our test series.

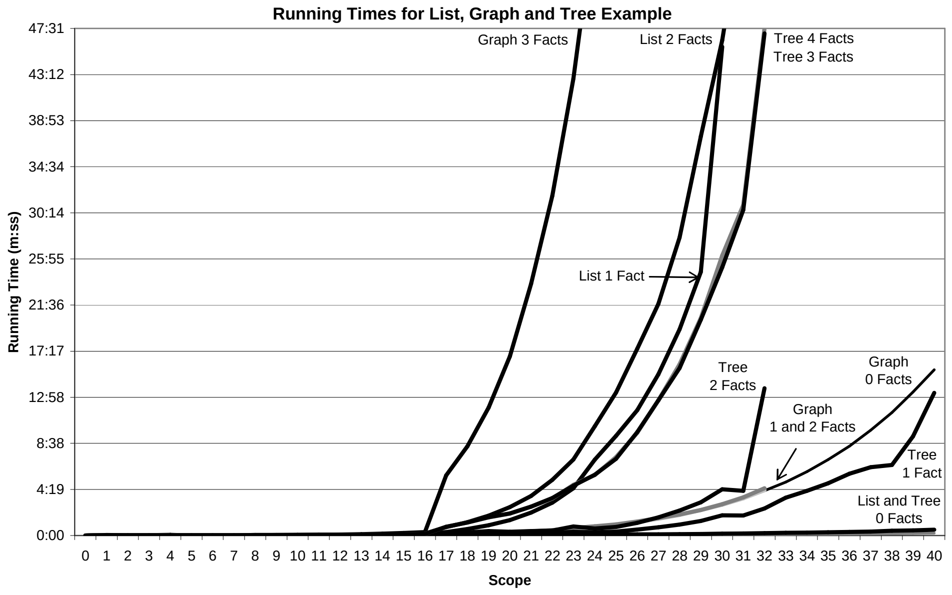

Figure 1: Comparison of conformance check running times for all three specifications, with differing numbers of facts. For the most part, specifications with no additional facts require less than a minute of running time. The *Graph* specification, with all three of its explicit facts takes the most running time. The *List* specification, with 1 or 2 explicit facts, is comparable to the *Tree* specification with 3 or 4 facts. Finally, the *Tree* specification with 1 or 2 facts and the *Graph* specification with 0, 1 or 2 facts all have comparable running times.

General comments results for each test family are summarized below:

list The conformance check of the *List* specification with no explicit facts takes well under a minute, regardless of the scope of analysis (up to 40). Increasing the number of explicit facts drastically increases the running time.

graph The running times of the conformance check of the *Graph* specification with 0, 1 or 2 explicit facts are almost identical, although the addition of the third and final fact drastically increases the running time. This specification is the only one with a ternary relation; the higher arity obviously affects the running time, as even the specification with no additional explicit facts takes longer than the other two specifications without explicit facts. The conformance check for 1 and 2 facts becomes unstable around $scope = 33$. The running time with 3 facts was halted when it surpassed one hour.

tree The conformance check of the *Tree* specification with no explicit facts takes well under a minute. Increasing the number of explicit facts drastically increases the running time. However, the running time for 4 facts is not much greater than that for 3 facts, even though the specification should be more complex. The conformance check for 2, 3 and 4 facts becomes unstable around $scope = 33$.

Figure 1 confirms that the running time of the conformance checking phase depends exponentially on the scope. Also, notice how the running time remains near zero until the scope reaches approximately 16 and then suddenly becomes exponential. We suspect that the deterioration in performance is due to the fact that the Alloy Analyzer has been optimized to deal with small scopes [5]. Note that Embee itself has not been particularly optimized; we were interested in creating a functional prototype, not necessarily an optimally performing one.

5.1.2 Instability

As mentioned above, several of the test series become unstable around $scope = 33$. In other words, the simple data structures created in our tests suddenly become nonconforming at this scope. We attempted to explore this supposed nonconformance with the Alloy Analyzer; however, we have not been successful in using the Edit Instance command with such a high scope.

5.2 Specific Results

The remainder of this section discusses these results in the context of our primary and secondary research objectives.

5.2.1 Primary Objective

As discussed in Section 3.1, our primary objective was to compare the running times required by Embee and by the Alloy Analyzer for Embee's conformance checking phase. The total running time for a conformance check is divided between Embee (creating and populating the dynamic mappings and extracting the appropriate Boolean array) and the Alloy Analyzer (interpreting the specification with the information from the Boolean array).

After the scope reaches 16, the bulk of the running time is required by the Alloy Analyzer to interpret the propositional formula with the truth assignment. According to our test results, when the scope of the analysis is less than or equal to 16, the Embee portion of Phase 3 takes between 15 and 90 percent of the total running time; however, the total running time of the phase remains less than a minute. However, as the scope increases past 16, the Alloy Analyzer suddenly accounts for 93 to 99 percent of the phase's running time. The running time after this point is significantly longer than for the smaller scopes.

Therefore, we conclude that Embee does in fact require more running time than the Analyzer to check the conformance up to and including $scope = 16$. However, the total running time of this phase is generally very short, making the division of labour between the two programs insignificant at small scopes. On the other hand, as the scope passes 16, the Alloy Analyzer suddenly requires much more time than Embee to check the conformance at a breakpoint. As the total running time of this phase becomes longer and longer, the fact that the Analyzer requires more than 90% of the running time becomes much more significant. Conformance checks at higher scopes are therefore limited by the speed of the Analyzer, and not by Embee itself.

5.2.2 Secondary Objectives

We also had three secondary objectives, as discussed in Section 3.2:

1. Our first secondary objective was to evaluate the effect of scope on running time of Phase 3. As can be seen in Figure 1, as scope increases, so does the time required to perform the conformance check. More importantly, however, is the fact that the running times of the conformance checks are negligible up to and including $scope = 16$. When the scope passes this number, the running time increases exponentially.

2. Our next objective was to evaluate the effect of explicit facts on the running time of Phase 3. The results of our tests, as shown in Figure 1, demonstrate that the addition of extra explicit facts to a particular specification occasionally increases the running time required to perform the conformance check. However, this is not always the case; for instance, the running times for the *Graph* specification are the same with 0, 1 and 2 facts. The only conclusion that we can infer from these results is that additional explicit facts may or may not increase the required running time. However, additional explicit facts will not decrease the required running time of the conformance check.

3. Our final secondary objective was concerned with whether or not simply counting the number of operators in the Boolean formula representing the specification was a possible indicator of the amount of running time required to perform a conformance check. Based on our estimates of the lengths of the formulas representing our specifications, we expected the *Graph* conformance checks to take the longest and the *List* conformance checks to take the least amount of time. It is indeed the case that the *Graph* specifications do require the most running time for the conformance check; this is thought to be caused by the fact that this is the only test family with a higher-than-binary relation. However, as can be seen in Figure 1, the *List* specifications actually require more time than the *Tree* specifications. In other words, simply counting the number of operations in the Boolean formula is <u>not</u> an accurate measure of how long the conformance check should take.

5.3 Surprising Results

For the most part, our results are as expected; Embee's runtime performance is acceptable for reasonably sized scopes, especially when compared with the running time required by the Alloy Analyzer during the third phase. However, two findings were surprising:

1. The first surprise was the fact that Embee's third phase takes less than a minute for any conformance check up to a scope of 16. This quick response occurs regardless of the complexity of the specification; all twelve of our test series performed this quickly. However, as soon as the scope of analysis passed 16, the running time of the conformance check increases dramatically, in some cases exponentially. As alluded to earlier, we believe that this threshold of 16 is somehow related to how the Alloy Analyzer is optimized to work in small scopes. We were not surprised that larger

scopes required significantly more time to analyze than smaller ones, but we were surprised by such an abrupt change in performance at *scope* = 16.

2. The second surprise is more disturbing. Most of the test series became unstable around *scope* = 33; in other words, the conformance check suddenly began to return false-negative results. In addition, the results were returned relatively quickly, implying that the entire Boolean formula was not being evaluated by the Analyzer before a false term was found. We are unsure as to why a conforming data structure with 32 nodes would suddenly become nonconforming when one more (appropriately placed) node was added. Obviously, this instability means that Embee should not be used when the scope of analysis becomes large; the results at such scopes could not be trusted.

6 Conclusions and Future Work

Based on the running time test discussed in this report, we are satisfied that the use of Embee to check conformance in certain scopes can be efficient. In fact, up to and including *scope* = 16, the conformance checking phase actually takes very little time. Unfortunately, the fact that many or our test series became unstable around *scope* = 33 means that Embee should not be used for large scopes.

6.1 Future Work

Future work will focus on the surprising results which we discovered. First, we would like to investigate why the Alloy Analyzer responds so quickly up to and including *scope* = 16. Is this threshold based on how the Analyzer has been optimized? Is it possible to increase this threshold? Second, we would like to explore the instability demonstrated around *scope* = 33. How is this threshold related to the Analyzer? Is this instability a bug in the Analyzer, or perhaps an unknown error in Embee's implementation? Has this type of problem been noticed by other users? Finally, we would like to experiment with other specifications and implementations to determine if these two thresholds hold over other test series.

A Alloy Specifications

The following specifications were used for the performance tests. The explicit facts have been indicated with comments.

A.1 *List*

```
module List

sig Node {
    next : option Node
}

sig List {
    first : Node
}

//Explicit Fact #1
fact NodeInOneList {
    all n : Node | one l : List | n in (l.first).*next
}

//Explicit Fact #2
fact NoCycle {
    all n : Node | n ! in n.^next
}

fun Show() {}
run Show for 3
```

A.2 *Graph*

```
module Graph

sig Node {}
{
//Explicit Fact #1
one g : Graph | this in (g.first).*(g.next)
}

sig Graph {
first : Node,
next : Node -> Node
}{
//Explicit Fact #2
//For every pair of nodes in a graph's next relationship,
//the left hand node (n1) must be in that graph
all n1, n2 : Node | n1 in n2.~next => n1 in first.*next
}
```

```
//Explicit Fact #3
fact NoCycle {
all g : Graph | all n : Node | n ! in n.^(g.next)
}

fun Show(){}
run Show for 3
```

A.3 *Tree*

```
module Tree

sig Key {}

sig Node {
    key : Key,
    left : option Node,
    right : option Node
}

sig Tree {
    root : Node
}

//Explicit Fact #1
fact KeysUnique {
    all t : Tree | all n1, n2 : nodesInTree(t) |
    n1.key = n2.key => n1 = n2
}

//Explicit Fact #2
fact EveryNodeInOneTree {
    all n : Node | one t : Tree | n in nodesInTree(t)
}

//Explicit Fact #3
fact NoCycles {
    all n : Node | n ! in descendants(n)
}

//Explicit Fact #4
fact OnlyOneParent {
    all n : Node | sole (n.~left + n.~right)
}
```

```
fun descendants (n : Node) : set Node {
    result = n.^(left + right)
}

fun nodesInTree(t : Tree) : set Node {
    result = t.root + descendants(t.root)
}

fun Show(){}
run Show for 3
```

References

[1] Michelle L. Crane. Runtime conformance checking of objects using Alloy. Master's thesis, School of Computing, Queen's University, 2003.

[2] Michelle L. Crane and Juergen Dingel. Runtime conformance checking of objects using Alloy. Submitted to RV'03, 2003.

[3] D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: The Alloy constraint analyzer. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 730–733. ACM Press, 2000.

[4] Daniel Jackson. Automating first-order relational logic. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 130–139. ACM Press, 2000.

[5] Daniel Jackson. Enforcing design constraints with object logic. In *Static Analysis Symposium*, pages 1–21, June/July 2000.

[6] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[7] Daniel Jackson. Micromodels of software: Lightweight modelling and analysis with Alloy. Technical report, Software Design Group, MIT Lab for Computer Science, February 2002.

[8] Sun Microsystems. Java^{TM}Platform Debugger Architecture. `http://java.sun.com/products/jpda`.

[9] Ilya Shlyakhter. Personal Communication, March 2003.