Electronic Notes in Theoretical Computer Science 89 No. 2 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume89.html 20 pages

Runtime Conformance Checking of Objects Using Alloy

Michelle L. Crane [1,2] Juergen Dingel [1,3]

Applied Formal Methods Group
School of Computing
Queen's University
Kingston, Canada

Abstract

Object models are an important part of most object-oriented software development methodologies, where they play a central role during the specification and design phases. However, their usefulness is much more limited during the implementation phase. In this paper, we demonstrate how confidence in source code can be increased by using runtime conformance checking to analyze the code with respect to an object model. More precisely, we use the Alloy Analyzer, developed at MIT, to determine automatically whether the runtime state of a program at certain user-specified locations conforms to a given object model. The design, implementation and evaluation of a prototype runtime conformance checker for Java programs with respect to Alloy object models is described.

1 Introduction

Object-oriented analysis and design have become very popular in recent years. Several modelling notations have been developed for this paradigm, with the current de facto standard being the Unified Modeling Language (UML) [3]. UML includes a rich set of artifacts, including object models (class diagrams), use cases, and message sequence diagrams. The Object Constraint Language (OCL) [20] extends UML by providing the ability to specify constraints that cannot be expressed in UML. Unfortunately, the semantics of UML and OCL are not completely formalized, which complicates the development of automatic analysis techniques and tools. In order to overcome these shortcomings,

[1] This research is supported by the Natural Sciences and Engineering Research Council of Canada under projects #228103-00 and #908-98.
[2] Email: crane@cs.queensu.ca
[3] Email: dingel@cs.queensu.ca

©2003 Published by Elsevier Science B. V.

researchers in the Software Design Group at the Massachusetts Institute of Technology (MIT) have developed Alloy. Alloy is a UML-compatible object modelling notation expressly designed with automatic analysis in mind.

An *object model* describes the objects that an object-oriented software system creates and uses during execution. It reveals the possible relationships between objects and expresses properties that they possess. The benefits of formally modelling software systems are well-documented [4,10,22]. By formalizing aspects of a system, developers are more likely to discover potential design flaws, as well as inconsistency, ambiguity, and incompleteness. Object models support the precise and concise description of central, high-level system properties. Consequently, they form an integral part of the specification and design stages of most object-oriented software development efforts.

Unfortunately, the usefulness of object models during later stages of the development cycle is currently much more limited—for two reasons:

- Although object models provide the implementer with a formal description of central properties, there are currently few techniques or tools that allow the implementer to check that newly developed or modified source code satisfies the constraints of the object model. In other words, the conformance of the actual source code with respect to the specified model cannot be automatically checked.
- Object models and code are typically presented as two separate and distinct artifacts. Currently, little tool support is available for keeping the object model and the source code synchronized; hence, it is difficult to ensure that changes in one are reflected in the other. Therefore, the model often becomes out-of-date or even obsolete and quickly ceases to be a useful development or documentation tool.

To address these issues, researchers have suggested combining the model and the code into one artifact [2,13]. This paper proposes to bridge the gap between models and code with a different approach—that of runtime conformance checking. To this end, we take advantage of the Alloy language and its associated analysis tool. More precisely, we present the design and implementation of a tool, called Embee, which captures the runtime state of a Java program at certain user-specified points and then uses the Alloy Analyzer to determine automatically whether or not the state conforms to the Alloy object model. The tool thus helps to increase confidence in the correctness of the model and the developed or modified code, and makes it easier keep the model and code in sync.

We proceed by briefly discussing the Alloy language and the Alloy Analyzer tool in the next section. Section 3 describes the Embee tool with a running example. Section 4 examines some related work and Section 5 concludes and outlines future work.

2 Alloy and the Alloy Analyzer

2.1 The Alloy Language

Alloy is an object-modelling language, developed by the Software Design Group at MIT [7,9], which combines attributes of Z [18], UML, and OCL. Alloy uses a small, intuitive, ASCII-only syntax. The semantics is based on first-order logic, sets, and relations, which are used to represent relationships between objects. Several characteristics distinguish Alloy from UML/OCL, including the following:

- Alloy provides transitive closure operations, which allow for the succinct expression of reachability properties such as the cycle-freedom of lists.
- Alloy models are analyzable. Alloy boasts a complete and formal semantics, making automatic analysis possible. Also, the language was developed along with an automatic analysis tool [11]. Models can be built incrementally, using the Alloy Analyzer for simulation and assertion checking.
- Alloy models are declarative. Properties and constraints describe a system's state; operations are specified by describing the relationships between the objects in the old and new state. A declarative language is well-suited to incremental modelling because complex specifications can be easily composed. On the other hand, OCL is not as declarative because it inherits the complexities of many programming language notions, such as type casting and non-terminating or undefined expressions.

```
sig Key {}                        sig Node {
                                      key : Key,
sig Tree {                            left : option Node,
    root : Node                       right : option Node
}                                 }{
                                      one t : Tree | this in nodesInTree(t)
                                      this ! in descendants(this)
                                  }

fact OnlyOneParent { all n : Node | sole (n.~left + n.~right) }

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

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

Fig. 1. Excerpt of Alloy object model for a binary tree

An excerpt from an Alloy object model of a binary tree is presented in Fig. 1. This object model consists of three *signatures*, which declare three disjoint sets of atoms—`Key`, `Node` and `Tree` atoms. The `Key` signature contains no fields and is used here to represent a primitive or string key value. The `Node` signature contains three binary relations. The `key` field, or relation, maps nodes to keys. The `left` and `right` relations map nodes to nodes. The `option` qualifier indicates that every node is related to zero or one other nodes with the `left` relation, and to zero or one other nodes with the `right` relation.

The `Node` signature is followed by a second set of braces containing invariants, which must always be true for all nodes. The first invariant, or fact, calls the `nodesInTree(t:Tree)` function to state that every node must be in exactly one tree; i.e., trees may not share nodes and nodes may not exist outside of a tree. The second invariant calls the `descendants(n:Node)` function to state that nodes may not be contained in the set of their own descendants; i.e., there may be no cycles in trees.

The `Tree` signature also contains a binary relation, `root`, which relates every tree to exactly one node; this signature has no attached invariants. There is one more explicit fact, `OnlyOneParent`, which uses the transpose [4] operator `~` to specify that for every node `n`, there may be a maximum of one node in the set containing the nodes which relate to `n` through the `left` or `right` relations. In other words, every node must have zero or one parents.

Finally, the object model contains two functions, which are used in the facts as described above. The first function, `descendants(n:Node)`, takes a node `n` as a parameter and returns the set of nodes that are descendants of `n`, i.e., children, grand-children, etc. It does this by making use of the transitive-closure [5] operator `^`, which in this case, returns the set of atoms found by following the union of the `left` or `right` relations one or more times. The second function, `nodesInTree(t:Tree)` takes a tree `t` as a parameter and returns the set of nodes which are reachable from the root of `t`; it simply returns the root itself, as well as the descendants of the root.

2.2 Automatic Analysis and the Alloy Analyzer

A key advantage of using Alloy as a modelling language is that object models can be analyzed fully automatically. Although first-order logic is undecidable, it is possible to analyze Alloy models by restricting the search space to a certain finite *scope*. The Alloy analysis uses this integer scope to translate the model into a propositional formula and employs a selection of off-the-shelf SAT solvers to determine whether or not there exists an *instance* within the scope that satisfies the formula [7]. An instance is a set of atoms and relationships between these atoms that conform to the structure and constraints defined by the signatures, relations and facts. The finite scope limits the number of atoms from any signature in the returned solution. In our example, a scope of 3 would mean that any satisfying instance of our object model would have no more than 3 atoms from the `List` signature and no more than 3 atoms from the `Node` signature.

There are two ways of using the analysis; the first is to check the consistency of a model, by finding an instance that satisfies it. This consistency check is essentially a simulation, or animation, of the model. The second type of analysis is to check the properties of the model. A desired prop-

[4] The transpose $\tilde{}r$ of a relation r is its mirror image, i.e., $\tilde{}r = \{(b,a)|(a,b) \in r\}$.

[5] Given a binary relation $r : A \to A$, the transitive closure $\hat{}r$ is the relation $\hat{}r = \bigcup_{i=1} r^i$.

erty is asserted and the analysis attempts to find a counter-example, i.e., an otherwise-satisfying instance of the model that does not satisfy the assertion. These two types of analysis are useful for determining if the model is under- or over-constrained and in supporting the incremental development of models [9].

The use of a finite scope makes the analysis decidable but also incomplete. If an instance satisfying the formula cannot be found within a certain scope, that does not imply that the model is unsatisfiable; an instance may be found if the scope is increased. Likewise, the lack of a counter-example for an assertion does not imply that the asserted property holds in a larger scope.

The Alloy Analyzer has been publicly available since September 1999 [11]. It provides a well-designed and intuitive graphical user interface, as well as command-line functionality. The Analyzer automates the analysis described above, permitting the user to edit a specification, analyze it, and examine the resulting instances or counter-examples.

The size of the search space of an analysis is exponential in the size of the scope. For instance, a binary relation in a scope of k has $2^{k\times k}$ possible values and a specification with only three relational state components in a scope of 3 has about a billion states [11]. However, in small scopes, the analysis is remarkably fast; in fact, the Analyzer has been developed especially to quickly process such small scope analyses. Empirical results demonstrate that specifications with the default scope of 3 can be analyzed in well under one minute [7].

3 The Embee Tool

Our Embee tool makes use of the Alloy Analyzer to automatically check the conformance of a program's execution, at particular points in that execution. We define the *runtime state* of a program to be the collection of objects and their relationships that exists at the top of the program's stack at a particular point in the program's execution. In other words, the runtime state contains those objects and relationships that are actually accessible at that point in the program. In essence, our process captures the runtime state of the program at user-specified breakpoints and translates it into an internal representation akin to a propositional truth assignment. This assignment, along with a propositional formula representing the object model, is passed to the Analyzer's SAT solver. The solver returns a value of TRUE or FALSE, based on whether or not the truth assignment satisfies the formula.

3.1 Preparation

The user develops an object model specification using Alloy. Our example uses the binary tree model shown in Fig. 1. In principle, conformance checking can be performed on any Java code purported to be an implementation of this object model. However, we do assume that each signature in the specification

has a corresponding Java class. Conformance checking could be performed if a signature has not been implemented; however, any constraints placed on that signature would mean that the execution was nonconforming.

Fig. 2 shows the UML class diagram of our candidate implementation of the specification in Fig. 1. We have actually implemented a binary search tree instead of a simple binary tree; it is possible to implement *more* than what is specified in the object model. In this case, the `BinaryTreeNode` class implements the `Node` signature and contains two additional attributes: `data`, which is of type `String` and `parent`, which refers to another `BinaryTreeNode`. It is possible, though not necessary, to suppress the information from these extra attributes so that they do not complicate or slow down the conformance checking process. The `BST` class implements the `Tree` signature, even though the `BST` adds additional constraints on the ordering of nodes in the tree. In addition, both of these classes have several methods, none of which have been specified in the object model. Finally, the `Key` signature is not implemented as a new class; we will simply make use of the `java.util.String` class.

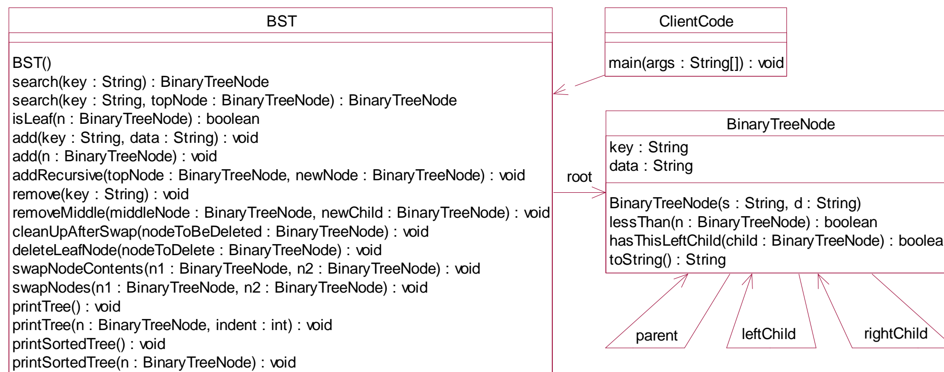


Fig. 2. UML class diagram of binary tree implementation

An implementation must be executed in order to check that execution for conformance. Therefore, the user must ensure that at least one of the implemented classes has a `main()` method; one of these executable classes becomes the *target* class. If there is no target class, then the user must create one. The target class should use the code that is to be checked for conformance; consider, for instance, the `ClientCode` class in Fig. 3.

Finally, the user creates a simple configuration file, which, at a minimum, contains the name of the target class and a list of desired breakpoints. An excerpt of the configuration file for our example is displayed in Fig. 4. It requests breakpoints at the beginning of line 11 of the `ClientCode` class and at the end of the `remove()` method in the `BST` class. The placement of breakpoints is not restricted to the target class; breakpoints could be placed at the beginning of any executable line of code, or at the end of any method, regardless of the class. In this case, the placement of the breakpoints occurs immediately before and after the `remove()` method is called in the target code.

```
1     public class ClientCode {
2         public static void main(String[] args) {
3             BST tree = new BST();
4             tree.add("H","hotel"); //becomes root
5             tree.add("D","delta");
6             tree.add("L","lima");
7             tree.add("B","bravo");
8             tree.add("F","foxtrot");
9             tree.add("J","juliet");
10            tree.add("N","november");
11            tree.remove("H");
12        }
13    }
```

Fig. 3. Implementation of target class to exercise `BinaryTreeNode` and `BST` classes

```
ClientCode
line : ClientCode : 11
method : BST : remove : (java.lang.String)
```

Fig. 4. Excerpt of configuration file, showing the name of the target class, as well as desired breakpoints

3.2 Embee Phases

The execution of Embee occurs in three distinct phases, as shown in Fig. 5.

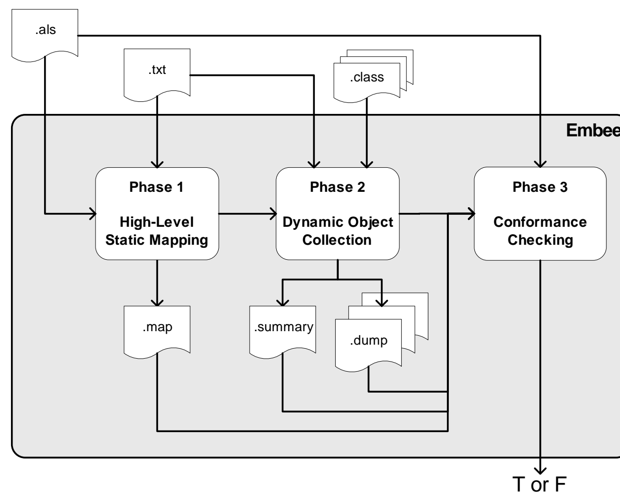


Fig. 5. High-level view of Embee execution

3.2.1 Phase 1: High-Level Static Mapping

In general, every signature in the object model should have a corresponding Java class; however, we do not require that the naming scheme of the Alloy model and the implementation be the same. The high-level static mapping is used to link signatures and relations in the model with the corresponding classes and attributes in the code. By default, Embee assumes that the specification signature and relation names correspond exactly to the implementation class and attribute names. The user may specify alternate signature-class and field-attribute mappings in the configuration file. If no mappings are speci-

fied, Phase 1 simply generates the default static mapping and presents it to the user, as shown in Fig. 6(a). If the default mapping is accurate, the user can continue with the Embee analysis. If not, the user can edit the mapping file and then continue. In our example, the `Tree` signature has been implemented as the `BST` class and the `Node` signature has been implemented as the `BinaryTreeNode` class, with different attribute names as well. The user would have to modify the mapping file as shown in Fig. 6(b). Alternatively, the user could include definition lines in the configuration file; Phase 1 would then be able to automatically generate the proper static mapping.

```
Node = Node
Node$key = Node.key
Node$left = Node.left
Node$right = Node.right
Key = Key
Tree = Tree
Tree$root = Tree.root
```

(a) Default static mapping

```
Node = BinaryTreeNode
Node$key = BinaryTreeNode.key
Node$left = BinaryTreeNode.leftChild
Node$right = BinaryTreeNode.rightChild
Key = String
Tree = BST
Tree$root = BST.root
```

(b) Modified static mapping

Fig. 6. Excerpt of high-level static mapping files, before and after user modification

3.2.2 Phase 2: Dynamic Object Collection

Once the high-level mapping is satisfactory, execution of the process continues with the second phase. The heart of this phase is a stand-alone Java program called *StateDumper*, which is responsible for executing the target, halting execution at the desired breakpoints, iterating through all of the objects that exist in the top of the target program's stack frame at each breakpoint, and outputting that information into a series of dump files, a sample of which is shown in Fig. 7. A summary file is also created, containing the names of all of the dump files, along with a count of the maximum number of objects of any one type at each breakpoint. This number will serve as the scope for the Alloy analysis. The Alloy Analyzer limits its search space to a maximum of *scope* atoms from each signature; it may be possible to find a satisfying instance with less than this number of atoms. Embee calculates the value of *scope* based on the number of objects accessible at a particular breakpoint; in this case, it is a count, not an upper limit.

The StateDumper program was developed using the Java Platform Debugger Architecture (JPDA), which is available from Sun Microsystems. The JPDA provides debugging support for the Java 2 platform, as well as infrastructure for the creation of end-user debugger applications [17]. Our use of this infrastructure is basic; the JPDA provides classes and methods that allow the StateDumper to execute the target in a second virtual machine, halt the execution at breakpoints, and retrieve information about objects at each breakpoint.

In order to make Embee as user-friendly as possible, we wanted to avoid forcing the user to instrument the implementation code; therefore, we de-

```
BreakpointEvent at Line 11 in method ClientCode.main(java.lang.String[])
...
Dumping: instance of BST(id=57)
- Field BinaryTreeNode BST.root = instance of BinaryTreeNode(id=54)
Dumping: instance of BinaryTreeNode(id=54)
- Field java.lang.String BinaryTreeNode.data = "hotel"
- Field java.lang.String BinaryTreeNode.key = "H"
- Field BinaryTreeNode BinaryTreeNode.leftChild = instance of BinaryTreeNode(id=58)
- Field BinaryTreeNode.parent = null
- Field BinaryTreeNode BinaryTreeNode.rightChild = instance of BinaryTreeNode(id=61)
...
```

Fig. 7. Excerpt of object state dump file for the line breakpoint

liberately avoided the requirement for any modification of the target code. All constraints are detailed in the Alloy specification, and the list of desired breakpoints is maintained in the configuration file. In the worst case, the user may have to insert a trivial executable line of code if none exists where a line breakpoint is desired.

3.2.3 Phase 3: Conformance Checking

Conformance checking is performed for each individual breakpoint. Information about the runtime state at that breakpoint is transformed into an internal representation akin to a propositional truth assignment. This transformation is not difficult; it starts by creating a set of objects representing every possible atom or relation between atoms that the Analyzer could create with the given object model. A dump file is then mined for information and a one-to-one mapping between possible Alloy atoms and actual Java objects is created; information about relationships between objects is also preserved. The final truth assignment representation is passed, along with the original object model, to the Alloy Analyzer. The Analyzer transforms the object model into a propositional formula, then uses a SAT solver to determine whether or not the truth assignment satisfies the formula. If so, then the objects at that particular breakpoint do indeed conform to the model. Fig. 8 contains an excerpt of the conformance-checking output. In our example, the `BinaryTreeNode` and `BST` objects created by the code conform only at the first breakpoint, i.e., before the root is deleted from the tree.

By the beginning of line 11 of the client code, a binary search tree with seven nodes has been created; at this point, the structure of the tree conforms to the constraints in the object model. Line 11 calls on the tree's `remove()`

```
...
LineBreakpointEvent.  Line: 11.  Class: ClientCode.  Method: main(java.lang.String[]).
Objects contained in file config_1.dump
Conformance at this breakpoint :)
...
MethodExitEvent.  Class:  BST.  Method: remove(java.lang.String).
Objects contained in file config_2.dump
NO CONFORMANCE AT THIS BREAKPOINT!!!
...
```

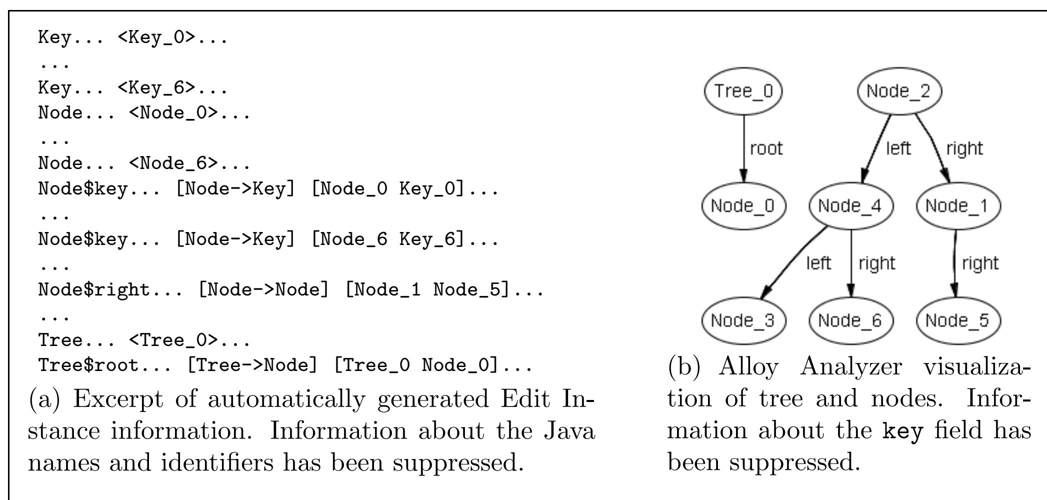
Fig. 8. Excerpt of Embee conformance-checking results

method, removing the root. At the end of this method however, the tree no longer conforms to the Alloy specification.

3.2.4 Past Embee: Exploring Nonconformance

It is possible to make use of the Alloy Analyzer in order to visualize and explore the nonconforming state. Embee is capable of automatically producing a list of all atoms and relations that represent an instance; an excerpt of this information for the second breakpoint is shown in Fig. 9(a). Each line contains details about either an atom/object or about a relationship between atoms/objects. The details refer to both the signature, field and atom names used by the Analyzer, and the Java classes, attributes and unique identifiers produced by Embee. This information can then be manually input into the Analyzer using its *Edit Instance* command.

The Analyzer will then display the propositional formula and highlight the nonconforming sections. The Analyzer's *Visualize* command can also be used to provide a graphical representation, as shown in Fig. 9(b).



(a) Excerpt of automatically generated Edit Instance information. Information about the Java names and identifiers has been suppressed.

(b) Alloy Analyzer visualization of tree and nodes. Information about the `key` field has been suppressed.

Fig. 9. Exploring the second, nonconforming breakpoint

Exploring the nonconforming instance with the Analyzer, we discover that a fact associated with the `Node` signature is not being met. The fact was `one t : Tree | this in nodesInTree(t)`, which is essentially a reachability constraint; i.e., every node must be reachable from the root of exactly one tree. As can be seen from the visualization in Fig. 9(b), only one node, `Node_0`, is actually reachable from the root of a tree. All of the other nodes form a binary tree, but are not reachable from any tree. In fact, the other nodes form the binary tree that would have resulted if the root node, `Node_0`, had been correctly deleted from the tree. It turns out that there was a typographic error in the `swapNodes()` method called by the `BST` class's `remove()` method. In this swap method, a temporary node was used to swap the deleted node with its successor, i.e., the next node in sorted order. If one of the nodes to be swapped was the root node, this error ensured that the root pointer was

never actually set to the new root (`Node_2`) of the tree. This is a subtle error; it only becomes apparent if the root of the tree is deleted.

3.3 Running Embee

The Embee process can be executed in three different modes: batch, runtime and critical runtime.

In *batch* mode, the target program is executed from start to finish, with a dump file being created at each breakpoint. When the execution has terminated, the runtime states logged in each dump file are checked for conformance, and the results are output to the user. This could be termed 'post-runtime' analysis and is best suited for programs that always terminate relatively quickly.

In *runtime* mode, the thread of execution is passed between the second and third phases. Once the runtime state has been dumped to a file, it is immediately checked for conformance. This is as close to true runtime analysis as is possible with our use of the JPDA, which must actually halt the target program in order to examine its state. The *critical runtime* mode is the same, except that the analysis will halt as soon as nonconformance is discovered.

3.4 Implementation and Performance

We implemented Embee in Java because this is the language of the Alloy Analyzer. The prototype consists of 59 classes with approximately 4 800 lines of code. Embee currently supports command-line operations only, with several flags to allow customization.

We have tested Embee with various object models and implementations, for example, linked lists, directed acyclic graphs and binary trees. Table 1 contains various performance measurements for three of our test cases. In each case, a data structure was created with 20 items, e.g., a linked list with 20 nodes, etc. Breakpoints were processed after each addition of a node [6]; at each successive breakpoint, the scope increases by one. The scope refers to the scope to be used with the Alloy Analyzer. It is not necessarily equivalent to the total number of all objects that have been created at any particular breakpoint. Instead, the scope is calculated by counting the maximum number of objects, of any particular type, which are accessible in the top of the execution's stack. For the purposes of our tests, however, we have ensured that all objects created thus far are available at the breakpoint, and the total number of objects is actually $scope + 1$, i.e., all *scope* nodes plus the enclosing data structure.

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 object model.

[6] In the case of the Graph tests, breakpoints were processed after the addition of each *edge*.

Table 1
Running times for each phase and total running time of Embee

Test Case			Running Time (m:ss)				
Object Model	Max Scope	Number of Breakpoints	Phase 1	Phase 2	Phase 3		Total
					First 16	Last 4	
List	20	20	0:07	0:32	0:12	06:39	07:30
Graph	20	19	0:07	1:27	0:35	44:10	46:19
Tree	20	20	0:04	1:20	0:21	06:04	07:49

The three examples differ in complexity. The *List* specification contains two signatures, two binary relations and two explicit facts. The *Graph* specification contains two signatures, three relations and three explicit facts. However, one of the relations is binary and the other is ternary; the higher arity severely affects the running time of Phase 3. Finally, the *Tree* specification contains three signatures, four binary relations and four explicit facts. We fully expected the *Graph* example to require the most running time, due to the higher-arity relation. The difference between the other two specifications is more minor; their overall running times are similar. The running time of the conformance check is obviously dependent upon both the complexity of the specification and the scope of the analysis.

Fig. 10 confirms that the running time of the conformance checking phase depends exponentially on the scope. Again, notice how the running time remains near zero until the scope reaches approximately 16 and then suddenly becomes exponential. We suspect that this disproportionate performance is due to the fact that the Alloy Analyzer has been optimized to deal with small scopes [8] and we are currently in the process of researching this possibility [5].

Further experimentation reveals that after the scope reaches 16, the bulk of the running time is required by the Alloy Analyzer to interpret the proposi-

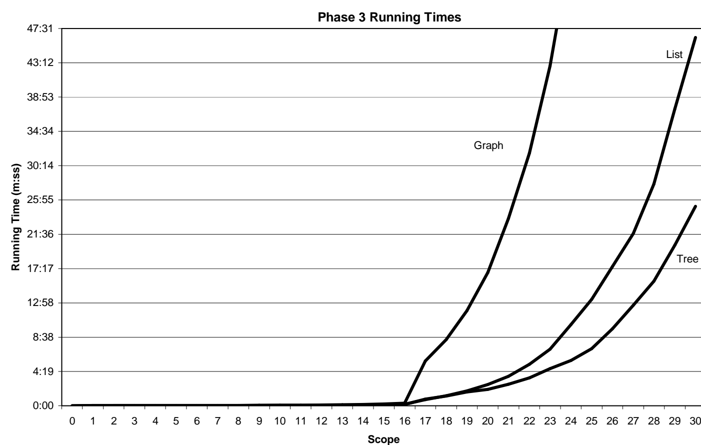


Fig. 10. Running times for conformance checking phase as scope increases

tional 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.

Since the overall analysis is exponential in the scope, Embee will typically be impractical for scopes greater than 20. Despite these limitations, we find our results encouraging. We believe that conformance checking of states with less than 20 objects per class can still be tremendously useful. Research with many Alloy models supports the *small scope hypothesis*, which states that many errors can be detected by considering only a small scope, such as 3 [8,10]. In addition, the majority of documented Alloy models are analyzed with scopes less than or equal to 6 [7]; in fact, the default scope is actually only 3 [8]. If even subtle errors can be found by using the Alloy Analyzer with such small scopes, it stands to reason that conformance checking with 20 or less objects per class will also be useful.

3.5 Code Analysis with Embee

3.5.1 Testing vs. Continuous Monitoring

Embee can be used for testing in either a *runtime* or *batch* mode. In this method of analysis, the target code would consist of a series of test cases, exercising whichever classes and methods interested the user. Breakpoints would be set at appropriate locations in the code, for instance, at the ends of important methods such as `add()` and `remove()`. Reporting of conformance would either occur during the execution or after the execution had terminated, depending on the mode. Every breakpoint where the runtime state conformed would be considered a successful test.

The tool can also be used for continuous runtime monitoring of the target code. In this case, the implemented code is simply exercised by normal operation. Reporting of conformance would occur throughout the execution, although in the *critical runtime* mode, execution would halt as soon as a nonconforming state was discovered.

3.5.2 Quality of the Analysis

There are two aspects that affect the quality of a conformance check: breakpoints and code coverage.

The number and placement of breakpoints are critical. More precisely, nonconforming states can easily escape our analysis, for instance, if the code does not contain enough breakpoints or if they are placed inappropriately. Consider for example, the implementation of a linked list with two methods, e.g., `add()` and `remove()`. The user might wish to verify that the `add()`

method preserves conformance. Suppose that `add()` preserves conformance, but `remove()` does not, and that the target code uses `add()` after `remove()`. A single breakpoint, at the exit of the `add()` method, would flag nonconformance correctly, but fail to indicate the true culprit. Currently, the user must determine where the execution should conform to its specification. Ideally, such points would be determined automatically, perhaps based upon some user input. However, we have not yet had the opportunity to pursue this avenue of research.

With respect to coverage, testing is inherently incomplete. Therefore, any testing environment needs to achieve a certain minimal level of data and control coverage to be able to deliver meaningful results. For example, the insertion of a breakpoint at the end of the faulty `remove()` method would not be useful if the test cases in the target code did not call that method.

3.5.3 Decidability and Completeness

As outlined in Section 2.2, Alloy's analysis consists of finding an instance that satisfies the model. This analysis is exponential in the size of the scope. Moreover, due to the bounded exploration of the search space, it is also incomplete [7,11]; that is, some of the Analyzer's results are somewhat inconclusive. On the other hand, Embee's analysis is much simpler. Checking that a given truth assignment satisfies a propositional formula is linear (in the size of the formula) and always terminates with a precise result. Our process benefits from the same 'theoretical curiosity' exploited by result-checking software: the computation to check if a given value constitutes a correct result is asymptotically less complex than the computation required to find the result [21].

3.6 Capabilities and Limitations of Embee Analysis

3.6.1 Capabilities

In principle, Embee is capable of checking the conformance of the target's execution with respect to any Alloy model; however, the following analyses appear to be the most useful: structural integrity, flawed specifications, number of objects and higher-arity relations.

The analysis is especially well-suited to confirming the integrity of data structures, such as lists, search trees, and directed acyclic graphs. Embee can be used to check conformance of implementations of operations on these structures. For example, as discussed in Section 3.2, it is possible to create an object model of a binary tree and then use Embee to confirm that the tree structure was maintained during execution of implemented operations and algorithms.

Although the Alloy Analyzer itself is a better tool for analyzing object models, Embee can highlight flawed specifications. The object model in our *Tree* example does not allow for trees to exist without any nodes; this would cause nonconformance at line 4 of our `ClientCode` class. In this case, the

nonconformance might indicate an error in the original object model; perhaps the specification should have allowed for empty trees.

It is possible to make some use of integers in Alloy; for example, it is possible to place an upper or lower bound on the number of a particular type of object. The constraint `fact Size{#Node > 1 && #Node < 5}` would mean that the number of `Node` objects encountered at a breakpoint would have to be greater than 1 and less than 5.

Alloy allows for higher-arity relations, such as the ternary relation in our *Graph* test case. Ternary and higher-arity relations are relatively intuitive in the modelling context. In addition, specific individual implementations of a higher-arity relation may also be simple. However, the gap between abstraction and implementation can be quite large, especially when trying to generalize how such relations could be implemented. We have found a method of representing higher-arity relations in Java implementations; however, it does force the user to implement such relations in a very specific manner. However, this restriction is counter-balanced by the fact that conformance checking of such relations can be performed.

3.6.2 Limitations

One of the major purposes of our research was to demonstrate that automatic conformance checking was feasible by implementing a working prototype. In doing so, we were forced to make some implementation decisions which unfortunately restrict the effectiveness of the end product.

For example, it is possible to specify an ordering on atoms in an Alloy specification, using Alloy's `std/ord` module. For example, it would be possible to specify that our binary tree is actually a binary search tree, i.e., that the key value of a node is greater than or equal to the key value of its left child, and less than that of its right child. Unfortunately, Embee currently cannot check the conformance of specifications that make use of this module; it should be possible to correct this limitation in the next version.

In addition, although the current implementation of Embee can inform the user that conformance has not been met at a specific breakpoint, it cannot provide any details as to why the runtime state is nonconforming. In the future, we would like to enable automatic visualization, obviating the need for the manual process described in Sec. 3.2.4.

Finally, the current implementation of the StateDumper and Embee programs requires that the target program be halted while its state is being output to file, and perhaps also during the conformance check (e.g., in runtime or critical runtime modes). It may be possible to make use of threads in order to allow the target to continue executing while a breakpoint is being handled. However, it seems apparent that while Embee may be used for runtime checking, it will probably not be useful for *real-time* runtime checking.

In addition to prototype-related restrictions, our analysis currently suffers from some fundamental limitations relating to: the level of abstraction,

primitive data types and checking operations.

Currently, the model and its implementation must exist on roughly the same level of abstraction. This restriction is unfortunate, given the fact that the very purpose of modelling is to focus on high-level aspects of a system and therefore to abstract away from implementation detail. For instance, a `set` in an Alloy signature could be implemented as an array, a `Vector`, a `List`, a `Set`, a `HashTable`, etc. in Java. Further research is required to explore to what extent our analysis can be extended to allow for the specification and code to exist on substantially different levels of abstraction.

Another limitation, also related to the level of abstraction, is influenced by the use of the Alloy language itself. Although Alloy contains no true built-in data types, such as integer or Boolean primitives, it is possible to make limited use of these primitives in object models. For example, the total number of atoms of a particular signature can be constrained and an ordering on a signature can be imposed with the use of the `std/ord` module. Likewise, Boolean types can be modelled with the `std/bool` module. In addition, it is simple to specify some constraints on such 'primitive' signatures, such as specifying that the value of a `key` field be unique. However, it remains difficult to specify, for instance, that this key must not exceed a certain integer value. The Alloy grammar allows for an `int` signature which seems to mimic the primitive integer in a programming language; however, we have not yet had success in creating valid object models employing this signature.

Embee is currently capable of single-state conformance checks; i.e., checking the conformance of a program at specific, individual points in its execution. Although it is possible to place breakpoints so that they occur immediately before and after an operation, it is not possible compare two breakpoints to determine how the program's state changes. In other words, Embee is not specifically designed to handle pre- and post-conditions on operations. Further research with the Alloy language and the Alloy Analyzer is required to determine whether or not expanding Embee's analysis to multi-state conformance checks would be possible.

4 Related Work

Several methods and tools are currently available that address the issue of comparing object-oriented programs against specifications.

4.1 Java Modeling Language (JML)

The Java Modeling Language is a notation that can be used to specify assertions about the detailed design of Java classes and interfaces, including pre- and post-conditions on methods [13,14]. A runtime assertion checker has been developed at Iowa State University that makes use of JML for runtime debugging and partial correctness checking [1]. The use of JML requires the user

to annotate the target code, i.e., the specification and code are combined in one artifact. Compilation of the code causes the annotated constraints to be expanded into runtime checks, that is, the conformance checking is performed as the code is executed, and not by a separate process.

4.2 Monitoring

The Monitoring and Checking (MaC) architecture, developed at the University of Pennsylvania, provides the ability to continuously monitor a target program's execution and to check the conformance of the execution against formal system requirements [12,15]. Users can formulate specifications using events and temporal operators. Java-MaC [12] is a prototype implementation of this architecture, designed for Java programs. Events are used to check whether the execution history conforms to the specification. The user needs to learn two specification languages, one each for low-level and high-level specifications, although the program itself is automatically instrumented and checked.

The Java PathExplorer (JPaX) is a separate tool developed at the National Aeronautics and Space Administration Ames Research Center [6]. It provides similar functionality to Java-MaC in that it has the ability to automatically instrument Java byte code and monitor program execution. The user needs to learn only one high-level logic language, which allows for the creation of temporal specifications. Conformance checking of an execution against high-level specifications is possible, as well as detection of low-level error conditions such as deadlock.

4.3 UML-Based Runtime Conformance Checking

Researchers at Queen's University are developing a tool that allows users to specify constraints on an object model using an extension of UML object diagrams. The user must create a set of Visual Constraint Diagrams (VCDs) [19], which represent nonconforming object states. In addition, runtime data snapshots of the target program are produced. A dynamic conformance checker compares the snapshots against the VCDs and outputs the results to the user. The main differences between this research and ours lie in the modelling language and the method of analysis. The UML-based approach depends on graphical specifications and uses a checking process based on the deductive database language GraphLog; our approach focusses on textual specifications and makes use of the Alloy Analyzer and its SAT solver.

4.4 TestEra

TestEra is a framework designed for the automated testing of Java programs, which also makes use of the Alloy Analyzer [16]. Alloy is used to describe the structural properties of a program's input, as well as correctness properties of

the program itself. The Analyzer is used to generate all non-isomorphic test cases based on the input's structure. TestEra then uses the target program to execute each test case, the results of which are translated back to Alloy and checked for conformance against the correctness properties. TestEra has been used to check interesting examples, including the naming scheme used by the Alloy Analyzer itself.

At first glance, TestEra and Embee seem very similar; both make use of Alloy and the Analyzer to check the conformance of Java programs, especially with respect to structural properties. However, there are several key differences. TestEra focuses on testing; Embee can be used for testing, but the original goal was to create a process that could be used for runtime conformance checking. TestEra is able to generate all non-isomorphic test cases in a certain finite scope; although the scope remains small, for instance, the examples all had $scope \leq 5$. In contrast, Embee does not generate test cases nor does it aim to check the correctness of the final result of a computation on all possible inputs. Rather, we want to ensure that certain user-specified intermediate states respect the invariant properties of the object model. In addition, we aim to surpass the small scope barrier, testing examples with $scope > 20$.

Finally, TestEra requires the manual creation of two translations, from Alloy to Java and vice versa. These translations are typically straightforward, but must be created for each program tested. On the other hand, with Embee, the user creates an appropriate object model in Alloy and includes breakpoints and definitions in the configuration file and Embee automatically performs the mapping between Java and Alloy. While this approach minimizes user involvement, it necessarily assumes a smaller gap between the specification and the program. In other words, the correspondence between the specification and implementation may be less restricted when using TestEra.

5 Conclusion and Future Work

Formal notations and tools for system design and analysis, such as the Alloy language and its automated Alloy Analyzer, provide developers with the ability to analyze and design object models. Unfortunately, once a model is actually implemented, there is no guarantee that the resulting program conforms to its specification. In addition, the fact that the model and implementation are separate artifacts, with little or no tool support, reduces the probability that the model will be maintained along with the code.

We propose to bridge the gap between object model specification and implemented code with a tool for runtime conformance checking. We have designed and implemented a prototype called Embee, which captures the runtime state of a Java program and then makes use of the Alloy Analyzer to determine whether or not this state conforms to an Alloy object model. Embee can help increase confidence in both the model and the correctness of the

implementation. In addition, the existence of such a tool may help convince developers to continue the maintenance of the object model after implementation has commenced. The effort of modelling can thus be amortized, not only over the analysis and design stages, but also over the later stages of software development.

Embee has been tested on medium-sized examples, i.e., with *scope* ≤ 30, such as the implementation of a linked list, directed acyclic graph, and binary tree. Our initial results are promising, especially with respect to the ability to check the conformance of higher-arity relations. Future work will focus on expanding Embee's capabilities to handle more complex object models, such as those making use of the `std/ord` and `std/bool` modules. In addition, we would like to improve the tool's reporting capabilities, for instance, by automatically linking back into the Alloy Analyzer in order to explore nonconformance visually. Finally, we are interested in simplifying the transformation process between Java objects and Alloy atoms, perhaps making use of automatic parser-generators.

Acknowledgements

We would like to acknowledge the invaluable assistance of Daniel Jackson and Ilya Shlyakhter from the Software Design Group at MIT. We would also like to thank David Lamb for his financial support.

References

[1] Bhorkar, A., *A run-time assertion checker for Java using JML*, Technical report, Department of Computer Science, Iowa State University (2000).

[2] Biagioni, E., R. Harper and P. Lee, *Implementing software architectures in Standard ML*, Position Paper (1994).

[3] Booch, G., J. Rumbaugh and I. Jacobson, "The Unified Modeling Language User Guide," Addison-Wesley, 1999.

[4] Clarke, E. M. and J. M. Wing., *Formal methods: State of the art and future directions*, ACM Computing Surveys **28** (1996), pp. 626–643.

[5] Crane, M. L. and J. Dingel, *Embee performance results*, Technical Report 2003-465, School of Computing, Queen's University (2003).
URL `http://www.cs.queensu.ca/TechReports/Reports/2003-465.pdf`

[6] Havelund, K. and G. Rosu, *Monitoring Java programs with Java PathExplorer*, Electronic Notes in Theoretical Computer Science **55** (2001), pp. 97–114.

[7] Jackson, D., *Automating first-order relational logic*, in: *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2000), pp. 130–139.

[8] Jackson, D., *Enforcing design constraints with object logic*, in: *Static Analysis Symposium*, 2000, pp. 1–21.

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

[10] Jackson, D., *Micromodels of software: Lightweight modelling and analysis with Alloy*, Technical report, Software Design Group, MIT Lab for Computer Science (2002).

[11] Jackson, D., I. Schechter and I. Shlyakhter, *Alcoa: The Alloy constraint analyzer*, in: *Proceedings of the 22nd International Conference on Software Engineering* (2000), pp. 730–733.

[12] Kim, M., S. Kannan, I. Lee, O. Sokolsky and M. Viswanathan, *Java-MaC: A run-time assurance tool for Java programs*, Electronic Notes in Theoretical Computer Science **55** (2001).

[13] Leavens, G., A. Baker and C. Ruby, *JML: A notation for detailed design*, in: *Behavioral Specifications of Businesses and Systems* (1999), pp. 175–188.

[14] Leavens, G., K. Leino, E. Poll, C. Ruby and B. Jacobs, *JML: Notations and tools supporting detailed design in Java*, in: *OOPSLA 2000 Companion*, Minneapolis, Minnesota, 2000, pp. 105–106.

[15] Lee, I., S. Kannan, M. Kim, O. Sokolsky and M. Viswanathan, *Runtime assurance based on formal specifications*, in: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.

[16] Marinov, D. and S. Khurshid, *TestEra: A novel framework for automated testing of Java programs*, in: *16th IEEE International Conference on Automated Software Engineering (ASE'01)*, 2001, pp. 22–32.

[17] Microsystems, S., *Java^{TM}Platform Debugger Architecture*.
URL `http://java.sun.com/products/jpda`

[18] Spivey, J., "The Z Notation: A Reference Manual," Prentice-Hall International (UK) Ltd., 1992.

[19] Turner, C., T. Graham, H. Stewart, C. Wolfe and A. Ryman, *Visual constraint diagrams: Runtime conformance checking of UML object models versus implementations* (2002), unpublished manuscript.

[20] Warmer, J. and A. Kleppe, "The Object Constraint Language: Precise Modeling with UML," Addison-Wesley Longman Publishing Co., Inc., 1999.

[21] Wasserman, H. and M. Blum, *Software reliability via run-time result-checking*, Journal of the ACM **44** (1997), pp. 826–849.

[22] Wing, J., *A specifier's introduction to formal methods*, Computer (1990), pp. 8–24.