Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 174 (2007) 99–114
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Model-driven Transformation-based Generation of Java Stress Tests

Victor L. Winter[1],[2]

Department of Computer Science
University of Nebraska at Omaha
USA

Abstract

This paper describes a practical application of transformation-based analysis and code generation. An overview is given of an approach for automatically constructing Java stress tests whose execution exercises all "interesting" class initialization sequence possibilities for a given class hierarchy.

Keywords: program transformation, strategic programming, Java class initialization, <`clinit`> method, JVM, TL, HATS

1 Overview

This paper describes a model-driven approach in which transformation can be used to automatically generate Java stress tests whose scale and complexity resist manual construction. The approach consists of a framework where a variety of Java entities can be modelled at various levels of abstraction. The models presented have structural properties that naturally lend themselves to transformation-based manipulation. In this setting, transformation-based analysis is performed on the most abstract form of a model and the goal of transformation-based generation is to derive a corresponding concrete model (i.e., a set of Java classes). All analysis and generation transformations discussed in this paper have been implemented in the higher-order transformation language TL [8] using the HATS system [7]. The resulting stress tests are being used to help validate that the SSP [6], a hardware implementation of a significant JVM subset, conforms to the specification of the Java Virtual Machine (JVM).

[1] This work was in part supported by the United States Department of Energy under Contract DE-AC04-94AL85000. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy.

[2] Email: `vwinter@mail.unomaha.edu`

doi:10.1016/j.entcs.2006.10.022

In the context of this paper, a *(stress) test* is the Java source code for a set of classes which, after compilation, can be given as "input" to an implementation of the JVM. The correct execution of this test program provides evidence that a certain portion of an implementation's behavior conforms to the JVM specification [3]. In particular, this paper focuses on the generation of stress tests that can be used to help validate the behavior of a JVM implementation with respect to class initialization. More specifically, we are interested in providing assurance that `<clinit>` methods are properly sequenced (i.e., are invoked at the proper time during program execution).

The remainder of this paper is as follows: Section 2 provides background on class initialization as it is specified for the JVM. Section 3 describes various models and model representations that are of interest to our testing goals. Section 4 discusses the selection, observation, and generation of tests. The section introduces the concept of a *discrimination net* to capture the notion of interesting clinit tests. Next, a design is given that enables `<clinit>` method sequencing to be observed in the context of a test program. This is followed by a discussion of transformation-based test generation. Section 5 discusses how clinit stress tests can be generated using the higher-order transformation language TL. Section 6 presents some results and section 7 concludes.

2 Background: Java Class Initialization

Class initialization is part of the linking phase of the JVM [3]. In Java, the initialization of a class takes place at most once during execution. We pick up the discussion of class initialization at the point where verification and preparation have already taken place. Furthermore, here we only describe the general case of initialization for user-defined classes. For example, we do not consider class initializations that are triggered as a result of the invocation of various reflective methods such as those that can be found in class `Class` or package `java.lang.reflect`. We also do not discuss initialization with respect to interfaces. And finally, due to space limitations, this background discussion does not cover the effects of *constant fields* and the *passive use* of classes on class initialization.

Generally speaking, class initialization involves executing the `<clinit>` method associated with a class. This method is generated by the compiler and contains code realizing all class variable initializers as well as static initializers. From an operational standpoint, class variable initializers and static initializers are executed in the order in which they syntactically appear in the class. The method `<clinit>` cannot be invoked directly at the source code level, but rather may only be invoked internally by the JVM in response to the *first active use* of a class.

There are three kinds of situations that constitute an active use of a class: (1) when a static field of a class is accessed, (2) when a static method of a class is invoked, and (3) when an instance of the class is created. At the bytecode level, there are four bytecodes whose execution constitute an active use of a class: `new`, `getstatic`, `putstatic`, and `invokestatic`.

The `<clinit>` method for a given class may be invoked at most once during the execution of a Java program. The internal structure of a `<clinit>` method (i.e., its body) is important to this discussion only to the extent that the body may contain an *active use* of another class.

The execution of the body of a `<clinit>` method for a class B should be suspended if the `<clinit>` method for the superclass of B has not been invoked [3]. The execution of a `<clinit>` method body should also be suspended if an attempt is made, in the method body, to evaluate an active use of a class whose `<clinit>` method has not yet been invoked. The execution of a suspended method must resume immediately after completion of the `<clinit>` method belonging to the class that triggered the suspension.

Rule 1:	A `<clinit>` method may be invoked at most once.
Rule 2:	The `<clinit>` method of a class B must be invoked before any active use of B may be evaluated.
Rule 3:	Before executing the body of the `<clinit>` method for a class B, the `<clinit>` method of the superclass of B must be invoked.
Rule 4:	The execution of the `<clinit>` method of B_i must be suspended upon encountering (in its body) an active use of a class B_k whose `<clinit>` method has not been invoked.
Rule 5:	The execution of a suspended `<clinit>` method (for class B_i) must resume immediately after completion of the `<clinit>` method (for class B_k) that caused the suspension.

Fig. 1. Class initialization rules

```
class A1                 { static int x = A2.w; }

class A2 extends A1 { static int w = 1;
                          static int x = A1.x; }
------------------------------------------------------------
class B1 { static int x = B2.w; }

class B2 { static int w = 1;
             static int x = B1.x; }
```

Fig. 2. The difference between Rule 3 and Rule 4

And finally, an active use of a class B may be evaluated at any time after the `<clinit>` method for B has been invoked (even though the `<clinit>` method for B may not be completed). This relaxation on the evaluation of active uses provides a straightforward means for resolving circular dependencies among `<clinit>` methods thereby ensuring that initialization sequences are well-founded.

[3] Initialization of an interface does not require initialization of its superinterface and consists only of executing the interface's initialization method.

Figure 1 gives a set of rules that are sufficient to assure the correctness of class initialization. Figure 2 highlights the distinction between initialization Rule 3 and initialization Rule 4. In particular, assuming A1 has not been initialized, a first active use of A2.x will result in A2.x being initialized with the value 0. In contrast, assuming B1 has not been initialized, a first active use of B2.x will result in B2.x being initialized with the value 1.

3 Concepts and Terminology

We use the term *model*, usually preceded by a descriptor (e.g., *class model*), to denote various Java entities. *Models* can have representations at various levels of abstraction – a characteristic that is exploited during transformation. There are two representational forms that are of particular interest: We use the term *abstract form* to refer to the most abstract representation of a model that we wish to consider. We use the term *concrete form* to refer to models represented in the syntax of Java that can be legally embedded within a particular Java source program, compiled, and executed.

The scope of our discourse ranges over the following models:

- *class hierarchy model* – This model represents a set of Java classes. In its abstract form, this model is represented as a list of abstract class models. In its concrete form, this model is represented as a list of concrete class models.
- *class model* – This model represents the clinit dependencies of a Java class including the dependency that exists between a class and its superclass. In its abstract form, this model is represented as a rewrite rule of the form:

  $$[B_1 \rightarrow B_2 B_3 \cdots B_n]$$

  where B_2 is the superclass of B_1 and $B_3 \cdots B_n$ denotes the active use sequence that occurs within the <`clinit`> method of B_1. The concrete form of this model is shown in Figure 4. The concrete form assumes that `<expression>` is the concrete form of an active use model corresponding to $B_3 \cdots B_n$.
- *active use model* – This model represents a sequence of active uses of a set of classes. In its abstract form, this model is represented as a list of class identifiers: $B_1 B_2 \cdots B_m$. In its concrete form, this model is represented as an expression of the form:

  $$(\text{B\_1.x} + \text{B\_2.x} + \cdots + \text{B\_m.x})$$

  where it is assumed that the classes $B_1, ..., B_m$ contain static declarations of the integer identifier `x`.
- *observed sequence model* – This model represents the clinit sequence that has been observed as a result of executing an active use model with respect to a given class hierarchy model. In its abstract form, this model is represented as a list of class identifiers: $B_1 B_2 \cdots B_k$. In its concrete form, this model is represented by

the class `observe` as shown in Figure 4.

- *initialization sequence model* – This model represents the order in which `<clinit>` methods should complete, according to the specification of the JVM, for a given *(class hierarchy model, active use model)* pair. In its abstract form, this model is represented as a list of class identifiers $B_1B_2\cdots B_r$. In its concrete form, this model is represented by the following boolean valued expression:

$$(\text{observe.B\_1} == 1 \;\&\&\; \text{observe.B\_2} == 2 \;\&\&\; \cdots \;\&\&\; \text{observe.B\_r} == \text{r})$$

  where it is assumed that an *observed sequence model* containing the classes $B_1B_2\cdots B_r$ exists.

4 Testing: Selection, Observation, and Generation

Our testing objective is to generate a Java source program that can be used to validate that an implementation of the JVM has behavioral properties that conform to the rules in Figure 1. Using the concepts defined in the previous section a *test*, in abstract form, is modelled as a tuple $(\mathcal{M}, a\_seq)$ consisting of a class hierarchy model $\mathcal{M}$ and an active use sequence a_seq. A *stress test* is modelled as a list of test models.

For a given class hierarchy model $\mathcal{M}$, it will generally be possible to construct an infinite set of active use sequences (e.g., B, BB, BBB, ...) and thus, an infinite number of tests $(\mathcal{M}, B)$, $(\mathcal{M}, BB)$, and so on. However, since $\mathcal{M}$ is finite, an argument can be made that there are only a finite number of "interesting" active use sequences. For example, one might argue that the active use sequence AA is redundant and therefore not interesting. Such arguments reflect assumptions, that are sometimes subtle, about the nature of the error that a test hopes to expose.

4.1 Selection: Discrimination Nets – Interesting Active Use Sequences

We say that an active use model is *complete* with respect to a given class hierarchy model $\mathcal{M}$ if and only if it guarantees the initialization, either directly or indirectly, of every class in $\mathcal{M}$. The abstract form of an active use model a_seq is *minimal* if (1) class identifiers occur at most once, and (2) a_seq does not contain a proper prefix that is complete. We refer to the set of all minimal active use models as a *discrimination net*. Figure 3 shows a class hierarchy and its discrimination net in graphical form. The abstract active use models belonging to the discrimination net are constructed by concatenating the class identifiers of all paths in the tree from root to leaf: $\{ABC, AC, BC, C\}$.

We are interested in the construction of stress tests that, for a given class hierarchy model $\mathcal{M}$, will validate all active use models belonging to the discrimination net of $\mathcal{M}$.

```
class A           { static int y = 0; }
class B extends A { static int y = 0; }
class C           { static int y = B.y; }
```

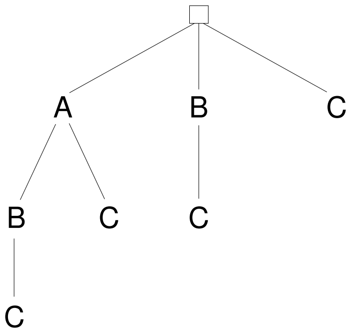

Fig. 3. A class hierarchy and its discrimination net

Class Model Template

```
class B_i [extends B_r] { static int x = <expression>;
                          static int pos = observe.setB_i(); }
```

Observed Sequence Model Template

```
class observe {
     static int B_1;
     static int setB_1() {   B_1 = next_position;
                             next_position += 1;
                             return B_1 ;
     }
     // declarations for remaining classes B_2, ... , B_n

     // the position counter used by all set methods
     static int next_position = 1;
}
```

Initialization Sequence Model Template

```
class set_and_check {
     static int a_seq = B_1.x + B_2.x; // Test sequence
     static void check() {
         System.out.println ( observe.B_j1 == 1 &&
                              observe.B_j2 == 2 &&
                              ...
                              observe.B_jn == n );
     }
}
```

Test Driver Template

```
class test {
     public static void main(String [ ] args) {
         set_and_check_1.check();
         set_and_check_2.check();
         ...
         set_and_check_m.check();
     }
};
```

Fig. 4. Concrete Form Templates

4.2 Observing <`clinit`> Method Sequencing

The clinit dependencies between classes can be seen as having a directed graph-like structure. These dependencies can be modelled by a class hierarchy model $\mathcal{M}$. Given a clinit dependency description $\mathcal{M}$, we are interested in creating, in concrete form, a hierarchy of classes $\{B_1, B_2, ..., B_n\}$ whose clinit dependencies correspond to $\mathcal{M}$.

A test class B_i belonging to a class hierarchy $\mathcal{M}$ will have a concrete form conforming to the class model template shown in Figure 4. The `extends B_r` portion of the class definition is optional and is only included when required by the dependency graph description $\mathcal{D}$. In the class `B_i`, the variable `x` is assigned to an expression whose evaluation explicitly triggers the sequence of active uses called for by $\mathcal{D}$. In this discussion, we will use an expression of the form $B_{j_1}.x + B_{j_2}.x + ... + B_{j_k}.x$ to trigger the active use sequence $\langle B_{j_1}, B_{j_2}, ..., B_{j_k}\rangle$. The second statement in class `B_i` is a static declaration of the variable `pos` whose value is assigned to `observe.setB_i()`. The value of this variable represents the position of the class' <`clinit`> method in the overall class initialization sequence for the class hierarchy. If `pos` has a value of zero it we conclude that the <`clinit`> method for the class has not been invoked (or did not complete).

For a given hierarchy of classes $\{B_1, B_2, ..., B_n\}$, we construct an associated class `observe`. The class `observe` consists of integer and method declarations whose purpose is to positionally record when the <`clinit`> methods of $\{B_1, B_2, ..., B_n\}$ are invoked. Note that the recording of the position of `B_i`'s <`clinit`> method is done external to `B_i` (i.e., within the class `observe`). This permits us to later query `B_i`'s position in a clinit sequence in a manner that does not itself constitute an active use of `B_i`.

The `observe` class contains an integer and method declaration corresponding to each class B_i in the class hierarchy and has a structure conforming to the observed sequence model template shown in Figure 4.

For each class hierarchy $\{B_1, B_2, ..., B_n\}$, there is also an associated class `set_and_check`. The first statement in `set_and_check` is a static declaration of a variable `a_seq` whose initializing expression consists of a specific initial (top-level) active use sequence to be tested. The second statement is a declaration of a method `check` which accesses the positional elements of the class `observe` to see if the initialization of classes $B_1, B_2, ..., B_n$ conforms to the correct class initialization sequence that results from the evaluation of a_seq. The class `set_and_check` has a structure conforming to the initialization sequence model template shown in Figure 4.

Since <`clinit`> methods are invoked only once, in any given execution run, a hierarchy of classes $\{B_1, B_2, ..., B_n\}$ can only be used to test the behavior a single active use sequence. However, a different active use sequence for a given (fixed) class hierarchy can be tested in the same execution run by making a copy of the class hierarchy as well as the associated `observe` and `set_and_check` classes. Such copies can be made using standard renaming techniques. In this manner a set of tests can be created and executed from a testsuite's `main` method by simply calling the `check` method of every instance of class `set_and_check` that has been created. The template for this driver method is shown in Figure 4. Note that the invocation `set_and_check_i.check()` will trigger the execution of the `<clinit>` method for `set_and_check_i` which will result in the initialization of the variable `set_and_check.a_seq`.

4.3 A Test Generator

Let $\mathcal{M}_A$ and $\mathcal{M}_C$ respectively denote the abstract and concrete forms of a class hierarchy model. Let $\{aseq_1, ..., aseq_n\}$ denote the discrimination net for $\mathcal{M}_A$. Let $iseq_j$ denote the (correct) class initialization sequence implied by $(\mathcal{M}_A, aseq_j)$, and let $(\mathcal{M}_{A_j}, aseq'_j)$ denote a consistent renaming of $(\mathcal{M}_A, aseq_j)$. Under these assumptions, the transformational steps that need to be performed can be summarized as shown in Figure 5.

(1) $\mathcal{M}_A \Rightarrow (\mathcal{M}_A, \{aseq_1, ..., aseq_n\})$
(2) $\Rightarrow \{(\mathcal{M}_{A_1}, aseq'_1), ..., (\mathcal{M}_{A_n}, aseq'_n)\}$
(3) $\Rightarrow \{(\mathcal{M}_{A_1}, aseq'_1, iseq'_1), ..., (\mathcal{M}_{A_n}, aseq'_n, iseq'_n)\}$
(4) $\Rightarrow \{(\mathcal{M}_{A_1}, observe_1, set\_check_1), ..., (\mathcal{M}_{A_n}, observe_n, set\_check_n)\}$
(5) $\Rightarrow \{test, (\mathcal{M}_{C_1}, observe_1, set\_check_1), ..., (\mathcal{M}_{C_n}, observe_n, set\_check_n)\}$

Fig. 5. A summary of transformational steps

In step (1), the abstract model $\mathcal{M}_A$ is used to construct the discrimination net $\{aseq_1, ..., aseq_n\}$. In step (2) tuples are constructed by pairing the abstract model with each element in the discrimination net and tuple elements are consistently renamed. In step (3) an analysis is performed on each tuple $(\mathcal{M}_{A_j}, aseq'_j)$ yielding the expected initialization sequence $iseq'_j$. In step (4) the pair $(aseq'_j, iseq'_j)$ is used to generate an instance of the class `set_and_check` and the model $\mathcal{M}_{A_j}$ is used to generate an instance of the class `observe`. And finally, in step (5) the models $\mathcal{M}_{A_j}$ are transformed into concrete form and the driver class `test` is added.

5 Transformation in Practice

Many of the transformational steps in the test generator are straightforward and, due to space considerations, the concrete details of their implementation are not presented. However, highlights of basic transformations are shown in Figure 6.

Aside from various standard transformational issues, there are three primary transformational problems that must be overcome when generating Java stress tests in the manner described in this paper. First, one needs to define transformations that are able to construct a discrimination net for a given model $\mathcal{M}_\mathcal{A}$. Second, one needs to define transformations that are able to construct the expected initialization sequence $iseq$ implied by a model/activation sequence pair $(\mathcal{M}_\mathcal{A}, aseq)$. And third, one must develop transformations that are able to consistently use identifier names across the stress test (e.g., calls in the `main` method to instances of

Class Model Transformation

$[B_1 \rightarrow B_2 B_3 \cdots B_m]$ $\Longrightarrow$ class B_1 extends B_2 { static int x = $B_3 \cdots B_m$; }

$\Longrightarrow$ class B_1 extends B_2 {

    static int x = (B_3.x + ... + B_m.x);

}

Initialization Sequence Transformation

$B_1 \cdots B_r$ $\Longrightarrow$ static void check() { $B_1 \cdots B_r$ }

$\Longrightarrow$ static void check() {

    System.out.println(observe.B_1==1 &&...&& observe.B_r==r);

}

Test Class Transformation

class set_and_check_1 { ... static void check() { ... } ... }

$\vdots \quad \vdots \quad \vdots \quad \vdots$

class set_and_check_s { ... static void check() { ... } ... }

$\Longrightarrow$

public static void main(String[] args) {

    set_and_check_1.check(); ... set_and_check_s.check();

}

Fig. 6. Transformation Highlights

`set_and_check.check()`). In the next section we give a brief overview of TL. This is then followed by a discussion of implementation details of two of these three problems: initialization sequence calculation and the consistent use of names.

5.1 Overview of TL

This section gives a brief overview of TL, a labelled conditional (higher-order) rewriting language supporting a variety of strategic operators and generic traversals. For a more detailed discussion of TL see [8]. In TL, parse trees are the "objects" that TL programs transform. Rewrite rules have the following form:

$$r : lhs \rightarrow s^n \; [\mathit{if\ condition}] \qquad (1)$$

In this example, r denotes the label of the rule, lhs denotes a *pattern* describing a tree structure, s^n denotes a *strategic expression* whose evaluation yields a strategy of order n, and *if condition* denotes an optional Boolean-valued condition consisting of one or more *match expressions* constructed using Boolean connectives.

A *pattern* is a notation for describing the parse tree structures that are being manipulated. This notation includes typed *variables* that are quantified over specific tree structure domains; E.g., $stmt[\![\; id_1 = 5 \;]\!]$ is a tree with root $stmt$ and leaves id_1, =, and 5. In this context, the subscripted variable id_1 denotes a typed variable quantified over the domain of all trees having id as their root node. In general, a pattern of the form $B[\![\alpha']\!]$ is structurally valid if and only if the derivation $B \overset{+}{\Rightarrow} \alpha$ is possible according to the grammar and α' is obtained from α by subscripting all nonterminals occurring in α.

A *strategic expression* is an expression whose evaluation yields a *strategy* having a particular order. In the framework of TL, a *pattern* is considered to be a strategy of order 0. A rewrite rule that transforms its input tree into another tree is considered to be a strategy of order 1 (i.e., a first-order rule). Let s^1 denote a first-order strategy. Then the rule $lhs \rightarrow s^1$ denotes a second-order strategy (e.g., s^2), and so on.

A *match expression* is a first-order match between two patterns. Let t_1 denote a pattern, possibly non-ground, and let t_2 denote a ground pattern. The expression $t_1 \ll t_2$ denotes a match expression and evaluates to *true* if and only if a substitution σ can be constructed so that $\sigma(t_1) = t_2$. One or more match expressions can be combined using the Boolean connectives { *and*, *or*, *not* } to form the *condition* of a rewrite rule.

A *combinator* is an operator defined on strategies. Two widely used combinators are: (1) left-to-right sequential composition (<;), and (2) left-biased conditional composition (<+). Let s_1 and s_2 denote two strategies. The expression s_1 <; s_2 denotes the left-to-right sequential composition of s_1 and s_2. When applied to a tree t, this strategy will first apply s_1 to t and then apply s_2 to the result. In contrast, the expression s_1 <+ s_2 denotes the left-biased conditional composition of s_1 and s_2. When applied to a tree t, the application of s_1 to t is attempted, and if that succeeds, the result is returned; otherwise, the result of the application of s_2 to t is returned. In TL, if neither s_1 or s_2 apply then t is returned unchanged. In other words, in TL, failure is treated as an identity. This is one characteristic that distinguishes TL from systems like Stratego [5] and Elan [1].

TL supports a variety of standard generic traversals such as *top-down left-to-right*, which in TL is denoted by the keyword TDL. TL also supports the definition and use of higher-order generic traversals. Informally, one can think of a higher-order traversal as mechanism for dynamically collecting a number of strategies and combining them to form a new strategy. A common higher-order traversal is one that traverses a tree in a TDL fashion, applies a higher-order strategy s^{n+1}, and composes the resulting order-n strategies using the <+ combinator. In TL, this traversal is denoted by the keyword *lcond_tdl*.

5.1.1 The Transient Combinator

The transient combinator is a very special combinator in TL. This combinator restricts a strategy so that it may be applied *at most once*. The "at most once" property is the hallmark of the *transient* combinator.

Transients open the door to *self-modifying* strategies. When using a traversal to apply a self-modifying strategy to a term, a different strategy may be applied to every term encountered during a traversal. For example, let $r1 : int_1 \rightarrow int[\![2]\!]$ denote a rule that rewrites an arbitrary integer to the value 2. If such a rule is applied to a term t in a top-down fashion, $TDL\{r1\}(t)$, all of the integers in the term will be rewritten to 2. Now consider the following self-modifying transient strategy $r2$:

$$\begin{aligned} r2 : & \, transient(int_1 \rightarrow int[\![1]\!]) <\!+ \\ & \, transient(int_1 \rightarrow int[\![2]\!]) <\!+ \\ & \, transient(int_1 \rightarrow int[\![3]\!]) \end{aligned}$$

When applied to a term t in a top-down fashion, $TDL\{r2\}(t)$, this strategy will rewrite the first integer encountered to 1, the second integer encountered to 2, and the third integer encountered to 3. All other integers will remain unchanged.

5.2 Initialization Sequence Calculation

The basic idea for determining the initialization sequence for a given class hierarchy and active use sequence is as follows: The initial active use sequence $list_{a\_seq}$ is treated as a stack where the top element, B_i, denotes the `<clinit>` method for the class that is currently active. A currently active `<clinit>` method, B_i, is "processed" by:

(i) **Mark:** B_i is marked as having been invoked. Conceptually, this is accomplished by a rewrite of the form $B_i \rightarrow [B_i]$.

(ii) **Suspend:** B_i is suspended. This is accomplished by pushing the active use model associated with B_i on top of the stack (via an append transformation).

(iii) **Remove:** The "mark and suspend" transformation used to accomplish steps 1 and 2 is removed so B_i cannot be invoked a second time. This removal is accomplished via the *transient* combinator.

(iv) **Cleanup:** A residual "cleanup" transformation is created that removes all unmarked instances of B_i.

```
model_list     ::= list ";" model_list | ϵ
list           ::= item [ "+" | "&&" ] list | ϵ
item           ::= class_model | ...
class_model    ::= class_id | marked_class | abstract_class | concrete_class | ...
class_id       ::= Id
marked_class   ::= "[" Id "]"
abstract_class ::= "[" Id "->" list "]"
concrete_class ::= "class" Id [ "extends" Id ] body
...
```

Fig. 7. An extended-BNF grammar fragment.

The TL strategy realizing the transformation described in the previous paragraph is shown in Figure 8. A context-free grammar fragment defining the syntactic structure of the model representations to be transformed is shown in Figure 7.

In Figure 8, the strategy *compute_iseq* is a higher-order strategy that, when given the abstract form of a class hierarchy model ($list_M$) and an active use sequence ($list_{a\_seq}$) will transform $list_{a\_seq}$ into its corresponding class initialization sequence. This transformation is achieved through the application of the strategy $rm\_obj <; process[list_M]$ to $list_{a\_seq}$ using the traversal *TDFIX* as shown in the following strategic expression:

$$TDFIX\{rm\_obj <; process[list_M]\}(list_{a\_seq}) \quad (2)$$

The generic traversal *TDFIX* is user-defined and, as it is used in *compute_iseq*, will perform a single top-down traversal over $list_{a\_seq}$ exhaustively applying the strategy $rm\_obj <; process[list_M]$ to every term encountered during the traversal.

```
def TDFIX s = TDL{ FIX{ s } }

append: ...

rm_obj: $list[\![OBJ\ list_2]\!] \rightarrow list_2$

mark: $abstract\_class[\![\ [Id_A \rightarrow list_1]\ ]\!] \rightarrow$
        $list[\![Id_A\ list_2]\!] \rightarrow (append[list_1](list[\![\ [Id_A]\ list_2]\!])))$

suspend: $abstract\_class[\![\ [Id_A \rightarrow list_1]\ ]\!] \rightarrow$
           $list[\![[Id_A]\ list_2]\!] \rightarrow (append[list_1](list[\![\ [Id_A]\ list_2]\!]))$

cleanup: $abstract\_class[\![\ [Id_A \rightarrow list_1]\ ]\!] \rightarrow list[\![Id_A\ list_2]\!] \rightarrow list_2$

mark_suspend_cleanup:
    $abstract\_class_A \rightarrow$
    (
        $transient(mark[abstract\_class_A] <;\ suspend[abstract\_class_A])$
        <+
        $cleanup[abstract\_class_A]$
    )

process: $lcond\_tdl\{mark\_suspend\_cleanup\}$

compute_iseq: $list_M \rightarrow list_{a\_seq} \rightarrow TDFIX\{rm\_obj <;\ process[list_M]\}(list_{a\_seq})$
```

Fig. 8. The TL strategies for determining class initialization sequence

(Note that the strategy *rm_obj* will remove all occurrences of OBJ from our initialization sequence model. The reason for this is that we do not model the <`clinit`> behavior of the class `Object`.)

As shown, *process* is a second-order strategy that, when applied to a class hierarchy model ($list_M$), will produce a strategy that models the clinit sequencing behavior of the <`clinit`> method for each class in $list_M$. More specifically, *process* accomplishes this by traversing $list_M$ and applying the strategy *mark_suspend_cleanup* to each abstract class encountered and composing the resulting first-order strategies using the <+ combinator.

5.3 The Consistent Use of Names

When generating a stress test the problem surrounding the consistent use of names is an instance of the *distributed data problem (DDP)* [8]. The DDP arises when a semantic relationship exists between terms that are syntactically unrelated. In practice, this means that information (e.g., identifier names to be referenced) must be explicitly transported between terms using a mechanism other than an encompassing match or unification. The parameterization of transformations is a standard approach that is often used to address the DDP. Other approaches include (1) the dynamic creation of rewrite rules and strategies in either a first-order [4] or higher-order setting [8], and (2) the fusion of term structures in which data and computations can be combined [2]. In the transformations described in this paper, most instances of the DDP have been avoided through (1) appropriate choice of identifier names, and (2) consistent use of a single model to derive various components of the stress test. For example, in the class `observe`, a monitoring identifier of type integer is declared corresponding to each class whose clinit behavior we want to observe. These monitoring variables are then referenced within the method `check` which is part of an accompanying the class set_and_check. Thus, a relationship exists between the declaration of monitoring identifiers in the class observe and their subsequent use in the method check. Naming consistency can be preserved in this case by simply choosing the names of monitoring identifiers so that they are syntactically identical to the identifier of the class they are intended to monitor (e.g., the monitoring identifer for class C will be C). Given this approach, it is straightforward to derive consistent versions of the classes observe and set_and_check from a model $\mathcal{M}_{A_k}$.

The set method within the class observe adds a slight wrinkle to our approach. Ideally, we would like to have a function for generating fresh identifier names that has the following input/output behavior:

$$\textit{special\_new(B\_k) = setB\_k}$$

A generator such as *special_new* could be used to generate method declarations in observe and corresponding method invocations in set_and_check. Note that there is nothing remarkable about the function *special_new* other than its ability to produce an identifier token from the concatenation of two other identifiers. In TL, such a function can be easily defined, placed within a user-defined library, and made accessible within a transformation. Similar functions can be created to generate instances of the classes observe and set_and_check (e.g., the identifier *observe* is concatenated with the first class identifer occurring within a class hierarchy model to create a unique instance of the class observe).

Another instance of the DDP arises from the need to invoke all check methods from within the main method of the driver class test. Here, TL admits a novel solution that makes use of associative matching at the token level. Specifically, TL allows identifier patterns to be constructed containing one or more occurrences of a wildcard denoted by the symbol $*$. The original motivation for providing this capability in TL was to facilitate the kinds of transformations that are performed during weaving in AOP environments – specifically, AspectJ pointcut descriptors. However, as we see here, this capability serves other purposes as well.

```
collect_set_and_check:
    $id_1 \rightarrow transient(list_1 \rightarrow list[\![id_1.check();\ list_1]\!])$
    if $id_1 \ll wild\_id[\![set\_and\_check*\ ]\!]$

make_test_class:
    $model\_list_1 \rightarrow model\_list[\![model_1;\ model\_list_1]\!]$
    if $list_1 \ll TDL\{lcond\_tdl\{collect\_set\_and\_check\}\ [model\_list_1]\}(list[\![]\!])$
    and $model_1 \ll model[\![class\ test\ \{public\ static\ void\ main(String[]args)\{list_1\}\}]\!]$
```

Fig. 9. The use of wildcard matching in the construction of a test driver.

Figure 9 shows the transformations used to construct the driver class test. In the strategy *make_test_class* the evaluation of the strategic expression

6 Results

$$lcond\_tdl\{collect\_set\_and\_check\}\ [model\_list_1] \tag{3}$$

Abstract Model (Input)	**Total Number of Classes**	**Number of Test Groups**
$[A \rightarrow OBJ]$	21	4
$[B \rightarrow A]$		
$[C \rightarrow B]$		
$[A \rightarrow OBJ\ B]$	85	12
$[B \rightarrow A]$		
$[C \rightarrow D\ A]$		
$[D \rightarrow A\ B\ C]$		
$[E \rightarrow C\ ]$		
$[A \rightarrow OBJ]$	505	63
$[B \rightarrow A\ C]$		
$[C \rightarrow OBJ\ A]$		
$[D \rightarrow OBJ]$		
$[E \rightarrow D\ B]$		
$[F \rightarrow A\ B\ C\ D\ E]$		
$[A \rightarrow OBJ]$	2401	240
$[B \rightarrow A\ C]$		
$[C \rightarrow OBJ\ A]$		
$[D \rightarrow OBJ\ B]$		
$[E \rightarrow D\ B]$		
$[F \rightarrow A\ B\ C\ D]$		
$[G \rightarrow OBJ\ A\ F]$		
$[H \rightarrow G\ F\ A]$		

Fig. 10. Metrics on Generated Stress Tests

will traverse the tests in $model\_list_1$ and produce an instance of the strategy $transient(list_1 \rightarrow list[\![id_1.check();\ list_1]\!])$ for each instance of the class `set_and_check` encountered. In the solution presented, instances of `set_and_check` have been created by concatenating a unique suffix to `set_and_check` (e.g., `set_and_check_01`). In the conditional portion of the rule *collect_set_and_check*, instances of `set_and_check` are recognized by the wildcard match $id_1 \ll$ *wild_id*⟦*set_and_check*∗ ⟧.

The resulting instances of $transient(list_1 \rightarrow list[\![id_1.check();\ list_1]\!])$ are used to populate the term *list*⟦ ⟧ – an empty list. This is accomplished using the the top-down traversal *TDL*. The resulting list is then placed in the body of the `main` method of the class `test`.

Figure 10 gives metrics on the stress tests generated for several abstract models which serve as the input to the test generator. In Section 4.1 we gave a rationale for restricting our attention in test generation to active use sequences that are *minimal*. This limits the number of tests that can be generated for a given class hierarchy. For example, a hierarchy containing n classes can have at most $n!$ tests in its discrimination net. This situation occurs when all classes in the hierarchy have an abstract model of the form $[B_i \rightarrow OBJ]$. At the other extreme, the smallest discrimination net will have size n. This situation occurs when the dependencies within the hierarchy are such that the first active use of any class will bring about the initialization of the all the other classes in the hierarchy.

7 Summary and Conclusion

Stress tests can provide a significant contribution to the assurance argument for a system. Oftentimes stress tests have size and complexity attributes that make their manual generation impractical. Two major challenges that one faces when automatically generating stress test are: (1) developing a systematic approach for selecting test cases, and (2) constructing certificates that can be used to automatically check the results of test cases. In this paper we have presented an approach for clinit test generation where the selection of test cases and the generation of certificates takes place on the abstract form of a model and the results are then transformed into concrete tests.

References

[1] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of elan. *Electr. Notes Theor. Comput. Sci.*, 15, 1998.

[2] R. Lämmel and J. Visser. A Strafunski Application Letter. In V. Dahl and P. Wadler, editors, *Proc. of Practical Aspects of Declarative Programming (PADL'03)*, volume 2562 of *LNCS*, pages 357–375. Springer-Verlag, Jan. 2003.

[3] T. Lindholm and F. Yellin, editors. *The Java Virtual Machine (Second Edition)*. Addison-Wesley, 1999.

[4] K. O. M. Bravenboer, A. van Dam and E. Visser. Program transformation with scoped dynamic rewrite rules. Technical Report UU-CS-2005-005, Institute of Information and Computing Sciences, Utrecht University, 2005.

[5] E. Visser, Z. el Abidine Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *ICFP '98: Proc. of the third ACM SIGPLAN international conference on Functional programming*, pages 13–26. ACM Press, 1998.

[6] G. L. Wickstrom, J. Davis, S. E. Morrison, S. Roach, and V. L. Winter. The SSP: An example of high-assurance system engineering. In *HASE 2004: The 8th IEEE International Symposium on High Assurance Systems Engineering*, 2004.

[7] V. Winter and J. Beranek. Program Transformation Using HATS. In *In proceedings of Generative and Transformational Techniques in Software Engineering (GTTSE)*, pages 95–111, 2005.

[8] V. Winter and M. Subramaniam. Dynamic Strategies, Transient Strategies, and the Distributed Data Problem. *Science of Computer Programming (Special Issue on Program Transformation)*, 52:165–212, 2004.