Automatic Validation of Transformation Rules for Java Verification against a Rewriting Semantics*

Wolfgang Ahrendt[1], Andreas Roth[2], and Ralf Sasse[3]

[1] Chalmers University of Technology, Göteborg, Sweden,
ahrendt@cs.chalmers.se
[2] Universität Karlsruhe, Germany,
aroth@ira.uka.de
[3] University of Illinois at Urbana-Champaign, USA,
rsasse@uiuc.edu

Abstract. This paper presents a methodology for automatically validating program transformation rules that are part of a calculus for Java source code verification. We target the Java Dynamic Logic calculus which is implemented in the interactive prover of the KeY system. As a basis for validation, we take an existing SOS style rewriting logic semantics for Java, formalized in the input language of the Maude system. That semantics is 'lifted' to cope with schematic programs like the ones appearing in program transformation rules. The rewriting theory is further extended to generate valid initial states for involved program fragments, and to check the final states for equivalence. The result is used in frequent validation runs over the relevant fragment of the calculus in the KeY system.

1 Introduction

In our work we relate two formal artifacts dealing with the programming language Java. The first is a *sequent calculus* for *Java Dynamic Logic* (JavaDL), a program logic for Java source code. This calculus [2] is implemented in the interactive prover of the KeY system [1]. The other artifact is a rewriting logic semantics [11, 10] for Java, written as a rewrite theory $\mathcal{R}_{Java}$ in the input language of the Maude system [5]. The objective of the work is to achieve an *automatic validation* of certain parts of the JavaDL calculus with respect to $\mathcal{R}_{Java}$, taking advantage of the executability of $\mathcal{R}_{Java}$.

The particular calculus rules we want to validate with this approach are *program transformation rules* of the form (cf. Sect. 2)

$$\frac{\Gamma \vdash \langle \Pi' \,\mathsf{rs}\rangle\ \phi, \Delta}{\Gamma \vdash \langle \Pi \,\mathsf{rs}\rangle\ \phi, \Delta} \tag{1}$$

* This research has been partly supported by STINT (The Swedish Foundation for International Cooperation in Research and Higher Education) and by the ONR Grant N00014-02-1-0715.

Roughly speaking, this proof rule replaces, in the beginning of a list of Java statements, a match of Π by the corresponding instance of Π'. (rs stands for the list of remaining statements.) Even if this appears as a very special case, a large and important part of the Java related rules of the JavaDL calculus (about 45%) is of exactly that kind! Note that the applicability of rules of this particular shape does not depend on the logical context, as Γ, Δ, and ϕ match arbitrary (lists of) formulae. Neither is the context affected by the rule application. The soundness of such a rule only depends on Π and Π'. Therefore, validating the rule reduces to showing *semantical equivalence* of Π and Π'.

It is important to note that one cannot simply 'run' $\mathcal{R}_{Java}$, in spite of its executability, on Π and Π'. The reason is that the statements in Π and Π' are not in plain Java syntax, but *schemata* for Java code. An example for a program transformation rule is

$$\frac{\Gamma \vdash \langle \mathsf{typeof(e)}\ \mathsf{v_1} = \mathsf{e};\ \mathsf{typeof(n)}\ \mathsf{v_2} = \mathsf{n};\ \mathsf{l} = \mathsf{v_1} * \mathsf{v_2};\ \mathsf{rs}\rangle\ \phi, \Delta}{\Gamma \vdash \langle \mathsf{l} = \mathsf{e} * \mathsf{n};\ \mathsf{rs}\rangle\ \phi, \Delta} \tag{2}$$

Here, l, e, n, rs, $\mathsf{v_1}$, and $\mathsf{v_2}$ are *schema variables*, matching certain syntactical categories (Sect. 2), and typeof delivers the static type of its argument. Comparing such schematic program fragments raises several issues.

First of all, $\mathcal{R}_{Java}$ is made for computing with *concrete* entities, like concrete memory locations, concrete (primitive) values, concrete object references, and so forth. It is an essential part of this work to have extended $\mathcal{R}_{Java}$ to a *lifted* Java semantics, $\mathcal{R}_{Java^{lift}}$, executing also *schematic*, i.e. *abstract*, Java code. Some central ingredients are the storage of *conditional values* in the memory, and parameterizing the values of abstract expressions by *snapshots* of the dynamic parts of the execution state. One can easily imagine that such an abstract execution would explode beyond feasibility if applied to longer program schemata. However, the pragmatics of program transformation rules (used for verification) make the considered program fragments short enough to keep the execution by $\mathcal{R}_{Java^{lift}}$ feasible.

Another issue is that the syntactical categories of schema variables, while sufficient for the proof rule, are not detailed enough to induce a unique execution by $\mathcal{R}_{Java}$, which for instance would need to distinguish between local variables and object fields as instances of l. This problem is addressed by the generation of all possible (and often very many) combinations.

One of the potential errors in a transformation rule is that certain instantiations are forgotten, namely those in which the instance of *different* schema variables *coincide*. The validation takes care of this by creating all possible unifying combinations of variables before checking for equivalence.

Besides our restriction to transformation rules, we are further constrained by the fact that $\mathcal{R}_{Java}$, in its current form, does not support all features of sequential Java. In spite of those restrictions, we could apply the automated validation to 56 rules, three of which turned out to be incorrect. We also discovered some errors in the semantics. As noted in [10], the whole process can be understood as a *mutual debugging*, which we consider very natural in a context where the ultimate reference (here the Java language specification [7]) is informal.

In general, what we needed for our purpose was a semantic formalism which is executable yet *abstract*. Rewriting logic, with its special support for associativity and commutativity, suited this purpose well. For instance, we need to represent a memory and all we know is that it maps a location `L` to a value `V`. The memory can be represented by `[L,V] rm`, with `rm` being a *constant* representing the arbitrary rest of the memory, and the juxtaposition with empty syntax being the *associative and commutative multiset union*, allowing us to abstract away from the concrete position of the location `L` in the memory. Such abstractions are heavily used in semantics formulated in a rewriting logic framework [10], where states are concrete but left hand sides of rewrite rules are abstract. We need abstraction even more, as in our *lifted* semantics even the states are abstract.

The paper is structured as follows. In the next two sections we present the two formalisms which we are concerned with: the program transformation rules of the JavaDL calculus (Sect. 2) and the rewriting logic which we use as basis for the validation (Sect. 3). Our approach to validate program transformation rules is then described in Sect. 4. In Sect. 5 we explain our lifting of the semantics. In Sect. 6, our implementation and experiences are sketched, before we conclude in Sect. 7 with a comparison to other approaches.

2 A Calculus for Java Source Code Verification

The KeY system aims at the deductive verification of sequential Java programs. The verification is based on a sequent calculus for JavaDL, which covers, among the propositional and first-order rules, full sequential Java[1].

Java Dynamic Logic (JavaDL) is a multi-modal logic, described in detail in [2]. For the purpose of this paper it is sufficient to state roughly that subformulae can be of the shapes $[\pi]\phi$ and $\langle\pi\rangle\phi$, where π is a sequence of Java statements and ϕ is again a formula. The intuitive meaning of $[\pi]\phi$ is that, if π terminates normally ϕ holds in the final state; $\langle\pi\rangle\phi$ means that π must terminate and afterwards ϕ must hold. The logic is closed under the usual first-order quantifiers and junctors, so the typical Hoare triple $\{\psi\}\pi\{\phi\}$ is formalized as $\psi \rightarrow [\pi]\phi$. In the following we only consider formulae with modality $\langle\cdot\rangle$, the other modality is treated exactly the same way.

Example 1. For local variables `i` and `j` of type `int`, the following JavaDL formula, which is valid in all states, says that after executing the piece of Java code in angled brackets `j * j` equals `i`:

$$\langle \texttt{i=(j=i)*(i++);} \rangle\ \texttt{j} * \texttt{j} \doteq \texttt{i} \tag{3}$$

The JavaDL calculus rules that work on sequents consisting of JavaDL formulae can be divided into the following categories:

[1] More precisely, the target language is JavaCard, but the calculus covers a larger fragment of Java which can be characterized as Java with exactly one thread and without garbage collection.

1. axiomatic program transformation rules,
2. axiomatic rules connecting the program and first order logic,
3. axiomatic first-order or theory specific rules,
4. derived rules, i.e. rules whose application could be simulated by applying a series of axiomatic rules,
5. axiomatic rules that apply state changes (*updates*) on first order formulae.

The basic concept behind the JavaDL calculus is the paradigm of *symbolic execution*. In order to resolve a formula $\langle \pi_1 \dots \pi_n \rangle \, \phi$ (with statements $\pi_1, \dots, \pi_n$), π_1 is taken into focus first. If it contains complex expressions, like `i=(j=i)*(i++);`, rules of group 1 transform it into less complex expressions, in our example to `int eval1=(j=i); int eval2=i++; i=eval1*eval2;`. Otherwise the state change of the first statement is, by applying rules of group 2, memorized as an *update* written in front of the modality. E.g., (3) is transformed—by several rule applications—into the equivalent formula $\mathcal{U} \, \langle\rangle \, \mathtt{j} * \mathtt{j} \doteq \mathtt{i}$ where $\mathcal{U} = \{\mathtt{i} := \mathtt{i} * \mathtt{i}, \mathtt{j} := \mathtt{i}\}$ is an update capturing the effect of the considered code as a *parallel* assignment to `i` and `j`. When code in a modality is completely worked off, rules of group 5 make the formula pure first order, by simplifying and executing the accumulated updates.

All of the rules from the groups 1 to 4 are implemented as *taclets* [3]. Taclets are representations of traditional rule schemes, but additionally have an operational meaning. Also, they embody a precise notion of schematic expressions. *This work is only concerned with taclets of group 1.* These taclets are mostly concerned with correctly reflecting the sophisticated evaluation order of complex Java expressions. Due to this non-trivial task and the sheer number (see Sect. 6) of rules of this kind, correctness checks are highly desired. In the sequel, we will detail only those parts of taclets which are relevant for this work.

A program transformation rule is written as a taclet as follows:

$$\mathsf{find}(\langle \Pi \, \mathsf{rs} \rangle \; \mathsf{b}) \;\; \mathsf{varcond}(\mathsf{new} \; T_1 \mathsf{v_1}, \dots, T_n \mathsf{v_n}) \;\; \mathsf{replacewith}(\langle \Pi' \, \mathsf{rs} \rangle \; \mathsf{b}) \tag{4}$$

where Π, Π' are (schematic) sequences of Java statements. We call taclets which comply with this shape *program transformation taclets (PTT)*. Intuitively, such taclets implement the concept of rewrite rules: when they are applied during proof construction, an occurrence of a formula $\langle \Pi \, \mathsf{rs} \rangle \phi$ is rewritten to $\langle \Pi' \, \mathsf{rs} \rangle \phi$. Π' may contain new program variables declared in the varcond section.

Example 2. This is a PTT:

$$\begin{array}{l} \mathsf{find}(\langle \mathsf{l} = \mathsf{e} * \mathsf{n}; \mathsf{rs} \rangle \; \mathsf{b}) \\ \mathsf{varcond}(\mathsf{new} \; \mathsf{typeof(e)} \; \mathsf{v_1}, \mathsf{typeof(n)} \; \mathsf{v_2}) \\ \mathsf{replacewith}(\langle \mathsf{typeof(e)} \; \mathsf{v_1} = \mathsf{e}; \;\; \mathsf{typeof(n)} \; \mathsf{v_2} = \mathsf{n}; \;\; \mathsf{l} = \mathsf{v_1} * \mathsf{v_2}; \mathsf{rs} \rangle \; \mathsf{b}) \end{array} \tag{5}$$

Traditionally, one would denote the represented sequent rule as (2). Note however, that—in contrast to that rule—the taclet is applicable on both sides of the sequent, and even on *sub*-formulae of sequent formulae. Most importantly however, side conditions on the instantiations of the rule schema are explicitly defined with taclets.

Table 1. Schema variable sorts and instantiations for *Example 3*

Schema variable sort	Conditions on instantiations ι	Schema var. in (5)	ι in (3)
Formula	ι is a formula	b	`j * j = i`
Expression	ι is an expression	e	`(j=i)`

Lefthandside	ι is a local variable *or* a field with either no prefix or a prefix not possibly causing side-effects	l
		v_1
		v_2
			`i`
			`eval1`
			`eval2`

NonSimpleExpression	ι is an expression but does not satisfy the *Lefthandside* condition and is not a (possibly negated) literal	n	`i++`
RemainingStatements	arbitrary sequence of statements	rs	(empty)

Clearly, a taclet must be interpreted as a *pattern*: For instance, (5) should be applicable *for all* formulae b, *for all* Java expressions l, e, n, v_1, v_2 which satisfy certain criteria, and *for all* sequences of Java statements rs. Expressions in taclets usually contain *schema variables* (printed in sans-serif here) to capture this need for genericity. When a taclet is applied, schema variables are *instantiated* with concrete expressions. Schema variables are assigned *conditions* and, in a special declaration section, *sorts*. Conditions and sorts determine which concrete expressions are legal instantiations for the schema variable. A taclet is *applicable* if there are legal and consistent instantiations of all the schema variables of the taclet. Table 1 gives an overview of the most important schema variable sorts. All terminology in this table refers to [7]. For PTTs, there is only the condition varcond(new $T_1 \mathsf{v}_1, \ldots, T_n \mathsf{v}_n$), which requires instances of $\mathsf{v}_1, \ldots, \mathsf{v}_n$ to be fresh and of the (Java) types $T_1, \ldots, T_n$.

Example 3. Consider Table 1. Let the schema variables of the taclet (5) be declared as shown in the third column. The instantiations in the last column satisfy the conditions imposed by the second column *and* by the varcond condition of (5). Thus, taclet (5) is applicable to formula (3).

Taclets can be applied in a proof through either user interaction or the automated deduction engine. The effect of an application of a PTT is quite intuitive: the occurrence in the formula matching the find part of the taclet is replaced by the instantiated version of the replacewith part.

There is another bit to make the description of PTTs complete: The typeof(·) construct provides taclets with the static types of (instantiated schematic) expressions. This *meta construct* [3] allows for introducing declarations into the results of taclet applications as the following example demonstrates.

Example 4. When the taclet (5) is applied to (3) the following formula results:

$$\langle \texttt{int eval1=(j=i); int eval2=i++; i=eval1*eval2;} \rangle \; \texttt{j} * \texttt{j} \doteq \texttt{i}$$

Because of the variable condition in (5) two new variables of type `int` have been introduced since the expressions `j=i` and `i++` are both of that type.

3 The Rewriting Logic Semantics of Java

In this section, we introduce the semantics we validate against, and the framework in which it is formalized.

3.1 Rewriting Logic and Maude

Rewriting logic [9] is the logical framework in which the semantics of Java we want to use is given. A (simplified) rewrite theory is a triple $(\Sigma, E, \mathcal{R})$ where (Σ, E) is an equational theory with the signature Σ of operations and sorts and the set E of equations, and $\mathcal{R}$ is a set of rewrite rules. The equations and rewrite rules can also be conditional. The rewrite rules are always used modulo the equations. A rewrite rule $t \Rightarrow t'$, with t and t' terms over the signature Σ, is an inference from a logical point of view while from a computational point of view it is a concurrent transition of states.

Maude [5] is a high performance implementation of rewriting logic. Equations in Maude theories are directed, have to be terminating and need to have the Church-Rosser property. In Maude we mostly work on multisets as data structures due to the possibility of using the internal associativity, commutativity and identity axioms which are declared as attributes for an operator.

3.2 The Maude Rewriting Semantics of Java $\mathcal{R}_{Java}$

The rewrite theory for Java semantics[2], called $\mathcal{R}_{Java}$ in the sequel, was developed by Feng Chen at the University of Illinois at Urbana-Champaign and presented in the paper [6]. This rewriting logic theory is given as an executable specification in Maude, thus it gives us a Java interpreter for free. The semantics uses continuation-passing style (CPS) to keep track of the code which is to be executed. Continuations can roughly be seen as an executable stack of statements which can be restored anytime. The semantics uses an explicit environment and memory model, i.e. variables are mapped to locations inside the environments and those locations are mapped to values in the memory. We call the whole state information, including the memory and environments, *configuration* from now on. As is usual within such rewriting logic specifications most rewrite rules and equations can be used locally and do not need to specify precisely the rest of the state in which they can be used. There is no documentation by the developers of this Java semantics but to get an impression on how it is structured we recommend the paper [11] where a (simpler) semantics for a CaML-like language has been developed in great detail. For a more general account of the design of such semantics, and on Maude as a semantic framework, see [10].

In Fig. 1 we present the configuration parts of $\mathcal{R}_{Java}$ in Maude-style notation. With `code1` and `code2` being code pieces which shall be executed sequentially, the continuation looks like (a): `k` wraps a continuation so it can later be used inside a multiset. The environment (b), wrapped by `e`, maps variable names

[2] This Maude theory can be downloaded from http://fsl.cs.uiuc.edu/javafan/

(a) Continuation: k(code1 −> code2 −> ...)
(b) Environment: e([X1, L1] [X2, L2] ...)
(c) Context: c(k(code1 −> code2 −> ...), e([X1,L1] [X2, L2] ...), o(currObj))
(d) Memory: m([L1, V1] [L2, V2] ...), n(I)
(e) Static env.: s(staticEnv)
(f) List of classes: cl(listOfClasses)

Fig. 1. Important parts of an $\mathcal{R}_{Java}$ configuration

Xi to locations Li. The continuation, environment, and additionally the current object currObj, constitute one part of the overall configuration, the context (c). Moreover, explicit memory (d) is needed, mapping locations Li to values Vi. The next free location in the memory is denoted by an integer I. Other parts of the configuration are the static environment staticEnv and the list listOfClasses of all classes, used for instance in method lookups.

These items (and a few more which we omit here) are put together under the run operator. Any such configuration can be executed by $\mathcal{R}_{Java}$.

```
run(c(k (...),  e (...),  o (...)),  m (...),  n (...),  s (...),  cl (...))
```

Note that the comma ',' here is a multiset-union operator, both inside run and inside c. As an example of a rewrite rule operating on such a configuration, we show the rule for writing to the memory:

```
    c(k(change(V, L) -> K), Cnt),
    m([L, V'] M)
=> c(k(K), Cnt),
    m([L, V] M)
```

In this rule the actual Java code has been evaluated long enough to have been reduced to change(V, L). K matches the rest of the continuation. The context Cnt matches the subset of all other components wrapped inside c, apart from the explicitly given k. In the memory at location L there is a value V' which is overwritten. The rest M of the memory remains unchanged and the change code has disappeared from the continuation after its execution.

3.3 Limitations of $\mathcal{R}_{Java}$ and Improvements

$\mathcal{R}_{Java}$ is a prototypic formalization of the Java semantics, and therefore has a couple of limitations, which restrict the number of transformation rules to which we can apply our approach (see Sect. 6). Some interesting Java features are not modeled, such as abrupt termination, switch, conditional expressions, method overloading, and static class initialization. Some other features were realized in an incomplete or faulty manner. During the realization of our approach, we fixed several of these shortcomings. Finally, we have added additional features to $\mathcal{R}_{Java}$ by introducing type checks for assignments and type casts. More on the improvements to the original $\mathcal{R}_{Java}$ can be found in [12].

4 Validating Program Transformation Rules

The style of semantics formalized in the rewriting logic framework partly builds on the tradition of *structural operational semantics* (SOS)[3]. One central paradigm is to include a 'still-to-be-executed' program in the *state* of execution which is modified as execution proceeds. In SOS, one notationally separates the program π from the rest of the state, by writing (π, s). Correspondingly, by $(\pi_0, s_0) \to (\pi_1, s_1)$ we mean that there is a number of steps after which the execution of the program π_0, when started in s_0, results in the program π_1 to be executed from state s_1.[4] A special case is $(\pi\pi_{rs}, s_0) \to (\pi_{rs}, s_1)$, where the second program π_{rs} (remaining statements) is a suffix of the first, and a certain number of execution steps will resolve π completely, while π_{rs} is still untouched.

Now, a transformation rule of the shape (1) (or a corresponding PTT (4)) is sound if the following holds for all programs π matching the schema Π, all programs π' matching the schema Π', all *arbitrary* programs π_{rs}, and all states s_0 being 'admissible' w.r.t. $\pi\pi_{rs}$ and $\pi'\pi_{rs}$: If $(\pi\pi_{rs}, s_0) \to (\pi_{rs}, s_1)$ and $(\pi'\pi_{rs}, s_0) \to (\pi_{rs}, s_1')$, then s_1 and s_1' are 'equivalent'. We defer a discussion of state equivalence to Sect. 5.3. A state is called *admissible* w.r.t. some programs if those programs can possibly be executed starting from this state. For instance, the state must, in its environment, map all variables in π to some locations, and in its memory, map all those locations to values.

The above statement is quantified over infinitely many programs π, π', π_{rs} and states s_0. The goal is, however, to have an *executable* criteria for the statement. In short, the idea is to define a *lifted* semantics, executing the *schematic* programs Π and Π' *directly*, working on *generic* states. With such a semantics at hand, the 'universally quantified' soundness criteria given above reduces to showing: If $(\Pi\,\mathtt{rs},\ \mathtt{s}_{\Pi,\Pi'}) \to (\mathtt{rs}, s)$ and $(\Pi'\,\mathtt{rs},\ \mathtt{s}_{\Pi,\Pi'}) \to (\mathtt{rs}, s')$, then s and s' are equivalent, where $\mathtt{s}_{\Pi,\Pi'}$ is the generic state being admissible w.r.t. Π and Π', and `rs` is a generic constant representing the 'remaining statements', not being executed. For instance, validating the PTT (5) (or equivalent the rule (2)) amounts to executing both

- `l = e * n;` and
- `typeof(e) v1 = e; typeof(n) v2 = n; l = v1 * v2;`

from the generic state admissible for both, and comparing the results.

The realization of this approach is elaborated in the next section.

5 Lifting the Semantics

In order to enable the execution of schematic code, we can first of all turn several less problematic schema variables into generic constants, allowing the rewrite rules to perform symbolic computation. This, together with the complication of meaningful typing, is discussed in Sect. 5.1. *Schematic expressions*, however,

[3] See [10] for the similarities and differences.

[4] The usage of $\to$ instead of $\xrightarrow{*}$ conforms with rewriting logic rather than with SOS.

require some extra effort. Instances of schematic expressions might have arbitrary side effects on the state, but we do not know which. Moreover, the same schematic expression can appear more than once in schematic code, with the different appearances having *different results* and *different side effects*. Therefore, evaluating schematic expressions requires extra constructs, which we introduce in Sect. 5.2. Problems concerning fresh variables introduced by PTTs are solved in Sect. 5.3, and in Sect. 5.4 we refine our analysis by nondeterministically *identifying different* schema variables.

5.1 Schema Variables versus Generic Constants

When preparing a piece of schematic code (like `l = e * n;`) for execution, we model *side-effect free schema variables* as *generic constants*, with the effect that the rules of the rewriting semantics will perform symbolic computation. Such a generic constant is a true constant only to rewriting logic, i.e. on a technical level. Intuitively, however, it acts as a representative of *any* fitting expression. By side-effect free schema variables, we mean those where instantiations are restricted to expressions which, by their syntactic nature, cannot possibly have a side-effect. Luckily, the taclet language provides this information, among other things, by *sorts* (which we have not spelled out in taclet (5), but indicated in the first column of Table 1). It is actually the very purpose of sorts in the taclet language, to *constrain* the *applicability* of taclets during proof construction. It is not surprising that, for the sound application of certain rules, it matters a lot whether or not side-effects can arise. Here, the needs of theorem proving match well with the needs of symbolic computation, where side-effects matter even more. In the example, `l` is of sort *Lefthandside*, a sort which happens to embody side-effect freeness. Therefore, `l` can in principle be turned into a generic constant.

Unfortunately, we also have to deal with a certain mismatch between program logic rules and symbolic computation via a rewrite semantics. The latter is, even if symbolic, yet more concrete. For instance, a schema variable of type *Lefthandside* can be instantiated with either of: a local variable, a static field, or a field of the current object. As the rewriting semantics executes these different possibilities each in a different way, our approach requires to test out all of them. As we usually have several schema variables in a taclet, all possible combinations must be checked in the validation. This leads to an explosion of combinations. Fortunately, programs in PTTs are by their very nature quite small, containing usually at most five schema variables, which is why this approach is feasible.

5.2 Computing with the Unknown

Even with the help of generic constants, $\mathcal{R}_{Java}$ per se does not provide means to 'execute' arbitrary unknown expressions possibly having side-effects, like those matching the sorts *Expression* or *NonSimpleExpression*. To be able to treat those, we lift $\mathcal{R}_{Java}$ to a *rewrite theory for schematic Java* ($\mathcal{R}_{Java^{lift}}$) as described in this section. First of all, we note that the same expression, when executed twice in different states, can have different side-effects and results. On the other hand,

when executed twice but starting in the same state, side-effects and result will be identical. Therefore we introduce *snapshots* of the state capturing those parts of the configuration which both side-effects and result can depend on. This allows to compare two states in which such an expression is executed, and to decide whether the side-effects and results of two evaluations are the same.

We demonstrate the concept of snapshots by an example configuration in Fig. 2.a. All the sans-serif typed elements are operators of the semantics whereas the others represent elements of the appropriate types.

(a) run(c(k(*Code*), e(*Localenv*), o(*Currentobject*)), m(*Memory*), n(*Nextfreememcounter*), s(*Staticenv*), cl(*Listofclasses*), nextSnapshot(*Natnextsnapcounter*), snapshots(*Snapshotlist*), ...)

(b) (snap(*Natnextsnapcounter*), c(e(*Localenv*), o(*Currentobject*)), m(*Memory*))

Fig. 2. An example configuration (a) and a fitting snapshot (b)

Fig. 2.a shows that we extend the structure of configurations by a *Snapshotlist* and a *Natnextsnapcounter* (syntactically wrapped by snapshots or nextSnapshot, respectively). The snapshot taken for this very configuration is depicted in Fig. 2.b. Its first element (snap(*Natnextsnapcounter*) in this case) acts as a name for the snapshot, to be used as a parameter elsewhere (see below). After such a snapshot is taken, it is added to the *Snapshotlist*, and the *Natnextsnapcounter* is incremented. Using snapshots, we can now represent the state-dependent evaluation of unknown expressions. For that, what remains is to model the effect of an arbitrary side-effect on the memory.

The side-effects of any expression can be viewed in the following way: a number n of memory locations $L_1, \ldots, L_n$ is updated with certain values $V_1, \ldots, V_n$. We however do not know any of $L_1, \ldots, L_n$ or $V_1, \ldots, V_n$, nor even the number n of affected memory locations. Therefore, when modeling the side-effects of a symbolic expression e, to be evaluated in a symbolic state s, we represent $L_1, \ldots, L_n$ by the *symbolic location list* LI(e, s), parameterized over e and s. Accordingly, $V_1, \ldots, V_n$ is represented by the *symbolic value list* VI(e, s). Furthermore, we actually do not use the full (symbolic) state for s, but only the name of the state's snapshot.

Now, when executing the so represented symbolic side-effects on the memory, we replace the value of *each* memory location with a 'kind of' conditional term, called *extended conditional value*. (Simple conditional terms are insufficient for this task.) Suppose that, before executing e, some particular symbolic memory location L holds the particular value V. The execution of e triggers that V is rewritten to the extended conditional value

L in LI(e, s) ?? VI(e, s) :: V

This construct represents the new value and has the following meaning: if L is a member of the list $\mathsf{LI}(e,\ s)$ then the resulting value is the corresponding element in the list $\mathsf{VI}(e,\ s)$. Otherwise the result is V, which was the old value. Note that this replacement is performed at *each* location/value pair in the memory, but everywhere using the according L and V.

Extended conditional values cannot be further evaluated (since expressions e are symbolic) but instead remain in the memory as they are, which is fine since we just aim at comparing two resulting states.

We illustrate the lifted semantics with the help of the following example:

Example 5. The following taclet is a slight variation of (5) but it is unsound since the order of evaluation is wrongly simulated:

$$\begin{aligned}&\mathsf{find}(\langle \mathsf{l} = \mathsf{e} * \mathsf{n};\rangle\ \mathsf{b})\\ &\mathsf{varcond}(\mathsf{new\ typeof(e)\ v_1}, \mathsf{typeof(n)\ v_2})\\ &\mathsf{replacewith}(\langle \mathsf{typeof(n)\ v_2 = n; typeof(e)\ v_1 = e; l = v_1 * v_2; rs}\rangle\ \mathsf{b})\end{aligned}$$

After processing both programs as described above, we end up with the following two values as memory contents at the location that l is mapped to. To simplify the presentation, we omit certain complications of purely syntactical kind here. i stands for the initial snapshot counter.

- (resultof e in snap(i)) * (resultof n in snap(i+1))
- (resultof e in snap(i+1)) * (resultof n in snap(i))

A further analysis of the snapshots with names snap(i) and snap(i+1), which could in principle be equal but are different in this case, finally reveals that the two considered programs are in fact different in result and side-effects. To better understand the actual side-effects, just imagine we had in our memory any other location, say, l1, with value v1. Executing both programs would then lead to replacing v1 by one of the following new values, respectively:

- l1 in LI(n, snap(i+1)) ?? VI(n, snap(i+1)) ::
  (l1 in LI(e, snap(i)) ?? VI(e, snap(i)) :: v1)
- l1 in LI(e, snap(i+1)) ?? VI(e, snap(i+1)) ::
  (l1 in LI(n, snap(i)) ?? VI(n, snap(i)) :: v1)

5.3 State Equivalence

Recall that, after 'running' $(\Pi\,\mathtt{rs},\ \mathtt{s}_{\Pi,\Pi'}) \to (\mathtt{rs}, s)$ and $(\Pi'\,\mathtt{rs},\ \mathtt{s}_{\Pi,\Pi'}) \to (\mathtt{rs}, s')$, we require s and s' to be *equivalent*. We now explain what we mean by that. The states s and s' are considered equivalent if they are *equal modulo new variables*. A variable is called *new* if it is introduced by the transformation, and thus *only* appears in Π', and is freshly declared therein. Examples of such new variables are $\mathsf{v_1}$ and $\mathsf{v_2}$ in rule (2) and PTT (5).

The need for an extended notion of equivalence is obvious: variables newly introduced in Π' appear in the configuration representing s', but not in the configuration representing s, which is why these configurations cannot possibly be entirely equal. However since new variables cannot appear in the remaining

code `rs`, they *could* just as well *be removed* before executing `rs`. This is however not what the semantics does, as it is not designed for being aware of variables appearing anymore or not. Instead, we realize a certain removal of new variables within the *'comparison modulo'* of resulting states. This is part of the *rewrite theory for validating transformation rules*, $\mathcal{R}_{Java^{valTransf}}$[5], which further extends $\mathcal{R}_{Java^{lift}}$.

To get a handle on when to perform the *comparison modulo* we use a new *marker*, the pause operator, to indicate where the 'interesting' part of the program (either of Π or Π') is over, with only some 'uninteresting' rest rs left. Note that the following rewriting logic rule, which triggers the comparison modulo, only matches continuations starting with pause:

```
    compareResultsModNewVars(run(c(k(pause -> K), context), state),
                                 run(c(k(pause -> K), context'), state'))
= compareResult(removeNewVarsLocs(run(c(k(pause -> K), context), state)),
                removeNewVarsLocs(run(c(k(pause -> K), context'), state')))

    compareResult(run(c(k(pause -> K), context), state) ,
                  run(c(k(pause -> K), context') , state') )
=       run(c(k(pause -> K), context), state)
  == run(c(k(pause -> K), context'), state')
```

First, the new variables are removed from environments and memories in the actual state and in the snapshots. The 'cleaned' resulting states are then compared using Maude's default equality check ==.

5.4 Identical Instantiation of Different Schema Variables

As mentioned in Sect. 1, it can easily be forgotten that, in situations where a PTT applies, different schema variables can match the same instantiation.

Stenzel [13] remarks that a transformation `x=y++;` $\rightsquigarrow$ `x=y; y=y+1;` is wrong since an assignment `x=x++;` leaves `x` unchanged, while `x=x; x=x+1;` increments `x` (according to [7]). Stenzel discovered the erroneous transformation, which was part of his calculus, by an 'on paper' verification of the rules. Remarkably, the calculus we investigate here carried the same error, in the form of the taclet:

$$\mathsf{find}(\langle \mathsf{l}_1\texttt{=}\mathsf{l}_2\texttt{++;}\ \mathsf{rs}\rangle\ \mathsf{b}) \quad \mathsf{replacewith}(\langle \mathsf{l}_1\texttt{=}\mathsf{l}_2\texttt{;}\ \mathsf{l}_2\texttt{=}\mathsf{l}_2\texttt{+1;}\ \mathsf{rs}\rangle\ \mathsf{b})$$

Our automatic validation detects errors of this kind by means of nondeterministic rewrite rules for the generation of configurations, and using the Maude support for exhaustively trying out all branches. In our example, l_1 and l_2 are identified on the one branch, and distinguished on the other. Note that the whole idea of 'running' a schema Π instead of its instances π (Sect. 4) would be unsound if we forced constants representing unknowns to be different.

[5] Available at http://i12www.ira.uka.de/~aroth/download/maude/.

6 Automated Validation and Results

Our approach to validate the PTTs of KeY is implemented as a completely automated process. It consists of two steps: (1) Using the taclet infrastructure of KeY, the code transformation of each PTT is extracted and Maude code is generated which triggers the generation of start configurations and (2) Maude builds the actual start configurations and executes them as input to $\mathcal{R}_{Java^{valTransf}}$.

In the first step two tasks are accomplished: The Java syntax of the PTTs is transformed to that used by $\mathcal{R}_{Java}$ (and $\mathcal{R}_{Java^{valTransf}}$), which slightly differs from the standard syntax. More importantly, schema variables are replaced by concrete generic constants as described in Sect. 5.1. Depending on the schema variable sort *several* start configurations are generated, each containing another generic constant. If there is more than one schema variable in the considered programs, *all combinations* of their generic instantiations are generated.

KeY currently contains around 210 PTTs (of around 480 Java related rules). We could not check all of them mainly because of the prototypic nature of the Maude Java semantics $\mathcal{R}_{Java}$ (Sect. 3.3) and because some (37) contain advanced meta constructs which capture program transformations not expressible by pure schematic means. Despite these restrictions, 56 PTTs are currently treatable.

Our checker identified three unsound taclets, one as reported in Sect. 5.4, one for the analog case of the decrement operation, and one which was caused by evaluating a side-effect twice. With the help of logging output, one could quite easily find out in which cases problems occurred. After correcting the three rules, we were able to validate all of the 56 PTTs. The runs are sufficiently fast (around 3 minutes), thus confirming our estimations from Sect. 5.1 that the combinatorial explosion of cases is irrelevant for our purposes. Our implementation is now already used in practice within the KeY project. Nightly runs ensure that accidentally introduced mistakes in the rules are detected as soon as possible.

7 Conclusions and Related Work

The described approach achieves a completely automated validation of program transformation rules of the JavaDL calculus against a semantics in rewriting logic, a high level declarative formalism. The validation machinery is almost entirely defined in rewriting logic itself. For the purpose of validating transformations, we exploited (a) the precise formalization of the JavaDL rule schemas as taclets and (b) the executability of the rewrite semantics. As a major contribution, we lifted the Java rewrite semantics to deal with schematic programs. Moreover, we solved the issues arising from a certain mismatch in the typing systems of both formalisms, from newly introduced variables, and from potentially identical instantiations of different schema variables.

There is extensive literature relating program logic calculi and language semantics. We restrict ourselves to works targeted at similarly complete calculi over similarly complex languages (which actually happens to further narrow down to calculi over Java only).

We start with work targeting the *same* calculus. [4] describes how a taclet-specific mechanism ensures the soundness of *derived* rules (group 4 in Sect. 2). It creates correctness proof obligations from taclets, rendered in the object logic. In contrast to our work on axiomatic transformation rules, the justification of derived rules does not involve a definition of the (Java) semantics. In that respect, what comes closer is the work of K. Trentelman [14] on three JavaDL rules of group 2 (which connect the program and the logic part of sequents). Those taclets are proven correct w.r.t. a formalization of Java in Isabelle/HOL, called *Bali*. The whole metatheory for relating both formalisms is explicitly formalized within Isabelle/HOL. The correctness proofs of the taclets are therefore completely formal, and machine checked, but require non-trivial interaction.

In the LOOP project [8], a denotational semantics of Java is formalized as a PVS theory. Java programs are compiled into semantical objects, and proofs are performed in the PVS theory directly. On top of that, a Hoare-style and a *wp* style calculus are formalized as a PVS theory, and verified against the semantics within PVS. As opposed to 'usual' Hoare-style or *wp* calculi, these ones work on the *semantical* objects, not on the syntax of Java.

In [13], K. Stenzel reports on an 'on paper' verification of his dynamic logic calculus for Java against a big-step semantics for Java he developed as well. He found three mistakes in the calculus, one of which was also present in two rules of the calculus we consider here (see Sect. 5.4). We profitted from that work in the sense that it made us aware of the identical-schema-variable-instantiations problem. As a result, our mechanism can (and did) detect mistakes which are of this nature.

Except from [4], all these approaches have in common that the rule verification is performed by interacting with a proof system, or even by hand. In contrast to this, our approach is *much more lightweight*, as the 'mental reasoning' which determines for instance our lifting of the semantics, is *not* captured by a *formal meta theory* of any kind, thereby gaining a lower level of certainty. On the other hand, we achieve a *fully automatic* validation of more than 50 rules though the used semantics does not cover all features of (sequential) Java yet. We will however need to investigate whether our 'lifting features' are already sufficient or need further extension when the coverage is extended.

Another future work is to weaken the now very restrictive form of transformation rules, to also cope with simple dependencies from the logical context of the programs. This would allow for handling certain *branching* rules as well.

We consider it a strength of the approach (and the same holds for [14]) that the two artifacts, calculus and semantics, are defined in very different formalisms, by different people, for different purposes. We believe that some of the certainty which we lose by not performing formal meta reasoning is regained by the different origins of the formalisms we use for cross-validation.

Acknowledgments

We would like to thank Richard Bubel for many discussions and valuable feedback throughout this work. Many thanks go to Wojciech Mostowski for several

valuable hints and to Steffen Schlager for commenting on an earlier version of this paper. We also would like to thank José Meseguer for inspiring discussions about our work, and putting it in a bigger context [10]. Finally, we thank the anonymous referees for very valuable feedback.

References

1. W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.
2. B. Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. P. Jensen, editors, *Java Card Workshop*, volume 2041 of *Lecture Notes in Computer Science*, pages 6–24. Springer, 2000.
3. B. Beckert, M. Giese, E. Habermalz, R. Hähnle, A. Roth, P. Rümmer, and S. Schlager. Taclets: A new paradigm for constructing interactive theorem provers. *Revista de la Real Academia de Ciencias Exactas, Físicas y Naturales, Serie A: Matemáticas (RACSAM)*, 98(1), 2004. Special Issue on Symbolic Computation in Logic and Artificial Intelligence.
4. R. Bubel, A. Roth, and P. Rümmer. Ensuring correctness of lightweight tactics for Java Card Dynamic Logic. In *Proceedings of Workshop on Logical Frameworks and Meta-Languages (LFM) at Second International Joint Conference on Automated Reasoning 2004*, pages 84–105, 2004.
5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude Manual*, April 2005. Available from http://maude.cs.uiuc.edu.
6. A. Farzan, F. Chen, J. Meseguer, and G. Roşu. Formal analysis of Java programs in JavaFAN. In R. Alur and D. Peled, editors, *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 501–505. Springer, 2004.
7. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.
8. B. Jacobs and E. Poll. Java program verification at Nijmegen: Developments and perspective. In K. Futatsugi, F. Mizoguchi, and N. Yonezaki, editors, *Software Security – Theories and Systems*, LNCS 3233, pages 134–153. Springer, 2004.
9. N. Martí-Oliet and J. Meseguer. Rewriting logic: roadmap and bibliography. *Theor. Comput. Sci.*, 285(2):121–154, 2002.
10. J. Meseguer and G. Roşu. The Rewriting Logic semantics project. In *Structural Operational Semantics, Proceedings of the SOS Workshop, Lisbon, Portugal, 2005*, ENTCS. Elsevier, 2005. to appear.
11. J. Meseguer and G. Roşu. Rewriting Logic semantics: From language specifications to formal analysis tools. In *Proceedings of the IJCAR'04, Cork, Ireland*, volume 3097, pages 1–44. Springer-Verlag LNCS, July 2004.
12. R. Sasse. Taclets vs. rewriting logic - relating semantics of Java. Technical Report in Computing Science No. 2005-16, Fakultät für Informatik, Universität Karlsruhe, Germany, May 2005.
13. K. Stenzel. A formally verified calculus for full Java Card. In C. Rattray, S. Maharaj, and C. Shankland, editors, *AMAST*, volume 3116 of *Lecture Notes in Computer Science*, pages 491–505. Springer, 2004.
14. K. Trentelman. Proving correctness of JavaCard DL taclets using Bali. In B. Aichernig and B. Beckert, editors, *Software Engineering and Formal Methods. 3rd IEEE International Conference, SEFM 2005, Koblenz, Germany, September 7–9, 2005, Proceedings*. IEEE Press, 2005.