Deductive Verification of RTSJ Programs

Christian Engel
Universität Karlsruhe
engelc@ira.uka.de

Abstract

The Real-Time Specification for Java (RTSJ) defines a region-based memory model with the capability to explicitely free memory regions (so called scoped memory areas). For preventing dangling references it introduces runtime checks, raising errors in the case of a failure, to restrict the creation of references between objects residing in different regions. This work presents how an existing dynamic logic calculus for Java is extended to handle RTSJ's memory model, thus enabling to prove the absence of failed runtime checks. This is crucial for giving safety guarantees for RTSJ applications or could allow the Java Virtual Machine (JVM) to skip these kind of runtime checks resulting in improved performance.

Keywords: real-time, formal verification, symbolic execution, dynamic logic, scoped memory.

1 Introduction

In recent years a trend to make Java suitable for safety critical applications could be observed. One of the main deficiencies concerning the suitability of Java for real-time programming is its memory management featuring a garbage collector. For real-time applications it is not acceptable to be interrupted by garbage collection arbitrarily often and for unbounded periods of time since this would lead to indeterministic performance of the application.

The Real-Time Specification for Java (RTSJ) [3] addresses this issue by providing memory areas which can explicitely be freed and which are thus not subject to garbage collection. This introduces of course the danger of dangling references. RTSJ uses runtime checks to prevent the creation of references that can turn into dangling references when a memory area is freed. A failed check raises a runtime error. This work presents an approach for statically verifying the absence of this kind of runtime errors which is especially relevant when employing RTSJ programs in safety critical systems. The presented approach is based on dynamic logic [6] and implemented in the KeY system [1] a theorem prover featuring a dynamic logic for a sequential subset (excluding for instance reflection) of Java.

The remainder of the paper is organized as follows: Section 2 briefly summarizes the necessary background. Section 3 describes the main contribution of this work, namely the extension of the JAVADL calculus for treating RTSJ's scoped memory model. Section 4 reviews related work and Section 5 contains a wrap-up of the presented approach and an outlook on future work.

2 Background

In the following, the basics of dynamic logic and RTSJ needed to understand this work are briefly summarized.

2.1 Dynamic Logic

First-order dynamic logic (DL)[6] extends first-order predicate logic by a modality $[p]$ for every program p of some imperative programming language. For two first order formulas ϕ and ψ and a legal program $\mathtt{p}$, the formula $\phi \rightarrow [\mathtt{p}]\psi$, for instance, is valid iff for every program state s satisfying ϕ the execution of $\mathtt{p}$ when started in s either (i) terminates in a state satisfying ψ or (ii) does not terminate. This matches the semantics of the *Hoare Triple* [7] $\{\phi\}p\{\psi\}$. Beside this example for a partial correctness specification, total correctness is expressible by the diamond modality ($\langle\rangle$). Accordingly, the semantics of $\phi \rightarrow \langle\mathtt{p}\rangle\psi$ is that $\mathtt{p}$ terminates when started in an arbitrary state satisfying ϕ and ψ holds in the corresponding post state.

Dynamic logic formulas are interpreted in *Kripke structures*. A Kripke structure is defined by a tuple (S, ρ), where S is the set of first order structures representing program states and ρ is a function mapping each program p to a *transition relation* $\rho(p) \subseteq S^2$ such that $(s_1, s_2) \in \rho(p)$ iff executing p in s_1 leads to s_2. All states $s \in S$ share the same universe $\mathbf{U}$. A DL formula φ is called *valid* if $s, \beta \models \varphi$ for all states s of every Kripke structure and all variable assignments β. Symbols that can be interpreted differently in different states (such as program variables) are called flexi-

ble, those whose interpretation remains the same in all states (such as predefined arithmetic operators) are called rigid.

The semantics of the underlying programming language, which is needed for performing symbolic execution, is encoded in the calculus rules of a sequent calculus. This calculus operates on proof trees whose nodes are sequents. A sequent $\Gamma \Rightarrow \Delta$, where Γ (the antecedent) and Δ (the succedent) are sets of DL formulas, is valid iff the formula

$$\bigwedge_{\gamma \in \Gamma} \gamma \rightarrow \bigvee_{\delta \in \Delta} \delta \tag{1}$$

is valid. Thus the formulas in the antecedent can also be thought of as assumptions we can use to prove the succedent to be true.

A sequent calculus rule

$$\frac{\Gamma_1 \Rightarrow \Delta_1 \quad \ldots \quad \Gamma_n \Rightarrow \Delta_n}{\Gamma \Rightarrow \Delta} \tag{2}$$

is correct if the validity of the premises (the sequents $\Gamma_i \Rightarrow \Delta_i$ with $1 \leq i \leq n$) implies the validity of the conclusion ($\Gamma \Rightarrow \Delta$). This means that a rule is basically applied bottom up: The conclusion is the sequent the rule is applied to and the premises are the result of the application. For proving the validity of a formula Φ we start with a sequent $\Rightarrow \Phi$.

Example 2.1 (Calculus Rules) *The rule for a conjunction in the succedent splits the proof branch it is applied to:*

$$\mathsf{andLeft}\ \frac{\Gamma \Rightarrow a,\ \Delta \quad \Gamma \Rightarrow b,\ \Delta}{\Gamma \Rightarrow a \wedge b,\ \Delta}$$

The sets Γ and Δ contain all other formulas possibly occurring in the regarded sequent that are not relevant for the rule. We say that a proof goal can be closed if it is an instance of an axiom (i.e. a sequent schema for which all of its instances are known to be valid). The axioms are:

$$\frac{*}{\Gamma, false \Rightarrow \Delta} \qquad \frac{*}{\Gamma \Rightarrow true, \Delta} \qquad \frac{*}{\Gamma, \Phi \Rightarrow \Phi, \Delta}$$

A proof tree is closed if each of its leaves is closed indicating validity of its root node.

Since the verification of Java programs has to deal with side effects of program execution resulting in changes of the program state, the dynamic logic for Java utilized in the KeY system, called JAVADL, provides a means, called state updates, to describe those state transitions. A state update is basically a lazy substitution which is not evaluated as long as it is applied to a modality. It is sufficient to consider simple updates of the form $\{loc := val\}$ here, where loc and val are terms. The semantics of the update $\{loc := val\}\varphi$ where φ is an arbitrary JAVADL formula matches the semantics of $\langle$`loc=val;`$\rangle\,\varphi$ iff loc and val are side effect free. Symbolic execution, as we consider it here, computes the effect of a program by compiling it stepwise (operating always on the first active statement of the executed program) into an update. A rule for executing an assignment `x=y;` where `x` and `y` are variables is given by

$$\frac{\Gamma \Rightarrow \{\mathcal{U}\}\{x := y\}\langle \mathtt{p} \rangle\, \varphi,\ \Delta}{\Gamma \Rightarrow \{\mathcal{U}\}\langle \mathtt{x=y;\ p} \rangle\, \varphi,\ \Delta}$$

where $\{\mathcal{U}\}$ is an update and `p` the remainder of the program.

KeY possesses a frontend for the Java Modeling Language (JML) [10] which was used to read JML specifications of the RTSJ API (see Sect. 3). JML specifications are compiled to dynamic logic when they are imported in KeY.

2.2 RTSJ

The RTSJ defines a region-based memory model [12] featuring, in addition to the classical Java heap memory, two novel kinds of memory regions [3]: *immortal memory* and *scoped memory*. Both are represented by Java classes, RTSJ entails no changes to Java on the language level. An immortal memory area, which is always a singleton, is never garbage collected and never freed during the lifetime of the application. In contrast, a scoped memory area, of which arbitrarily many can be created by an application, is freed at well defined occasions, namely as soon as no thread is active inside it any more. Scoped memory areas (we call them just scopes in the remainder of the paper) can be entered and left by threads through the scope's `enter` method provided by the RTSJ API which also permits nesting of scopes. Outer (relative to the currently active scope from which memory is allocated by the `new` operator) scopes can be reentered using the outer scope's `executeInArea` method which does not change the nesting structure but only moves to another active scope. Since other scopes can be entered while executing in such an outer scope the nesting hierarchy of scopes is a cactus stack. Heap memory is in principle also usable by RTSJ based applications which is for afore mentioned reasons not advisable in a hard real-time context. This work will thus only consider programs running exclusively in immortal and scoped memory.

When assigning to locations of non-primitive type (that are no local variables) runtime checks are performed to prevent the creation of such references that can potentially give rise to dangling references. A failed check raises an `IllegalAssignmentError`. These checks enforce that for each reference `x.r` or `x[i]` to an object `y` (i) `y` resides in immortal memory or (ii) `y` resides in a scope s_1 and `x` resides in the same or an inner scope s_2 of s_1 (s_2 is located above s_1 on at least one branch of the scope stack). In short, the creation of references outliving the referenced object is avoided by this approach.

RTSJ also imposes certain well-formedness constraints on the scope stack. The so-called single parent rule ensures that each occurrence of a scope s_1 on the scope (cactus) stack has the same parent s_2. Since this also holds for s_2 and all preceding scopes as well the entire branch below s_1 is identical for each occurrence (on the stack) of s_1.

Restrictions: In this work we will only consider RTSJ programs complying to the following restrictions: We forbid (i) the usage of heap memory and (ii) the reentering of immortal memory using its `enter` method; accessing it via `executeInArea` is permitted however. This ensures that the only place a non-scope memory area can occur on the scope stack is its root. Even though this rules out legal RTSJ programs it is much less restrictive than safety critical Java profiles imposing similar restrictions [11, 8].

3 Calculus

The extensions to the existing JAVADL calculus required to support the scoped memory model can be subdivided as follows: (i) The semantics of the relevant parts of the RTSJ API needs to be described which is done by a reference implementation augmented with a formal specification mainly consisting of JML invariants. (ii) The scope stack is modeled by an abstract Java class whose semantics is described by invariants and method specifications. (iii) The nesting relation of scopes is represented by the partial order $\preceq$ and (iv) calculus rules for the symbolic execution of RTSJ programs are defined.

3.1 The RTSJ API and the Scope Stack

We model the scope stack by immutable instances of the Java class `MemoryStack`. Each of these instances only represents a sub branch of the entire cactus stack and is as such just a "normal" stack. Each scope is augmented with an attribute `stack` representing the subbranch of the cactus stack ending with the considered memory scope and starting with the immortal memory at the root of the stack. This is possible, as for every scope s the branch below each occurrence of s on the cactus stack is identical (see Sect. 2.2). If `stack` is `null` the scope is not located on the stack.

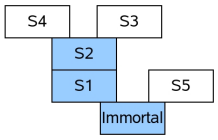

Figure 1. `S2.stack`

```
public abstract class ScopedMemory ...{...
  public /*@nullable@*/ MemoryStack stack;
  ...
  public void enter(){
    ...
    if(stack==null)
      stack=<currentScope>.stack.push(this); ...
```

Among other things (enforcing the single parent rule, increasing and decreasing the reference count of the scope, etc.) the `enter` method takes care that `stack` is correctly initialized. If a scope is entered and not yet contained on the stack, its attribute `stack` is set to `<currentScope>.stack.push(this)`, which does not change `<currentScope>.stack` since it is immutable, but only constructs a new stack from it with the newly entered scope on the top.

A lightweight way of formalizing this behavior of the stack is to define a sub stack relation $\preceq$, where $a \preceq b$ is true if a is a prefix of b. The formal specification of `push` then just has to express that the result is a newly created `MemoryStack` and that `s` $\preceq$ `s.push(a)` holds for every stack `s` and every scope `a`. The advantage of this approach is that, due to the immutability of `MemoryStack`, $\preceq$ can be rigid (its arguments, however, can be flexible) which makes it less involved to handle in dynamic logic than it would be the case for a flexible symbol. However, the `stack` attribute of a scope can be set to a different value during the program run. This means that each instance of `MemoryStack` is only a snapshot of a part of the scope stack taken at a certain time. In state s, a is an outer scope of b iff $s \models a.stack \preceq b.stack$.

3.2 Axiomatization of $\preceq$

By what has been described so far $\preceq$ is merely an uninterpreted predicate, so we have to axiomatize what it should semantically stand for. This is done via calculus rules as exemplarily demonstrated in the following. The relation $\preceq$ is a partial order and thus, for instance, antisymmetric:

$$\frac{\Gamma,\ o_1 \preceq o_2,\ o_2 \preceq o_1,\ o_1 = o_2 \Rightarrow \Delta}{\Gamma,\ o_1 \preceq o_2,\ o_2 \preceq o_1 \Rightarrow \Delta}$$

We declare a ghost field `ma` in class object for storing the memory scope the containing object is allocated in. There are a number of constraints on references enforced by the RTSJ that we can assume to hold in every reachable program state. The following rule states that in every reachable program state all non-static attributes, that are not `null`, point to the same or an outer scope:

$$\frac{\Gamma,\ o.a.\texttt{ma.stack} \preceq o.\texttt{ma.stack},\ \mathrm{RS} \Rightarrow o.a = \texttt{null},\ \Delta}{\Gamma,\ \mathrm{RS} \Rightarrow o.a = \texttt{null},\ \Delta}$$

where RS is a predicate that holds in exactly those states which are reachable by a legal RTSJ program. Thus we know that all axioms defining what a reachable program state is hold iff RS holds. Other rules define, for instance, the relation between immortal and scoped memory. Altogether, 18 other rules in addition the the ones shown above were needed, of which some are, however, redundant and merely defined for efficiency reasons.

3.3 Rules for Symbolic Execution

For the evaluation of RTSJ programs by symbolic execution we have to take into account the RTSJ specific runtime checks. This is exemplified by the rule handling an attribute write access (where o is a variable and v is a side effect free reference type expression):

$$\frac{\Gamma,\ \{\mathcal{U}\}(o \neq \texttt{null} \wedge \psi) \Rightarrow \{\mathcal{U}\}\{o.a := v\}\langle \texttt{p} \rangle\, \varphi,\ \Delta \quad \Gamma,\ \{\mathcal{U}\}o = \texttt{null} \Rightarrow \{\mathcal{U}\}\langle \texttt{NPE;}\ \ \texttt{p} \rangle\, \varphi,\ \Delta \quad \Gamma,\ \{\mathcal{U}\}(o \neq \texttt{null} \wedge \neg\psi) \Rightarrow \{\mathcal{U}\}\langle \texttt{IAE;}\ \ \texttt{p} \rangle\, \varphi,\ \Delta}{\Gamma \Rightarrow \{\mathcal{U}\}\langle \texttt{o.a=v;}\ \ \texttt{p} \rangle\, \varphi,\ \Delta}$$

where $\psi := v = \texttt{null} \vee v\texttt{.ma.stack} \preceq o\texttt{.ma.stack}$ states that v is either `null` or allocated in o's scope or an outer scope of it. In the above rule we have to distinguish three cases (indicated by the three premises): (i) o is not `null` in the state described by the update $\mathcal{U}$ and the assignment is legal with respect to the RTSJ constraints on this, (ii) $o = \texttt{null}$ holds and a `NullPointerException` is raised (abbreviated by `NPE;`) or (iii) the former two case do not hold and an `IllegalAssignmentError` (abbreviated by `IAE;`) is thrown.

4 Related Work

The presented work focuses on verifying real-time Java programs complying (with only minor restrictions; see Sect. 2.2) with the existing RTSJ. Most related approaches try to improve analyzability of real-time Java applications by further restricting or changing the RTSJ memory model losing, in turn, some of the flexibility it provides.

Kwon and Wellings [9] describe a memory management model making use of implicitly created memory scopes associated with each method leading to something comparable to stack allocation of objects only locally used by a method. The absence of explicite scope identities, for instance, eliminates the need for enforcing the single parent rule since it is impossible to reenter a scope. `IllegalAssignemntErrors` still remain an issue to consider, but checking their absence statically is eased by the simpler memory model and can for instance be done by escape analysis [5].

To reduce the error-proneness of RTSJ programs several profiles for safety critical Java (SCJ) applications have been proposed [8, 11] building upon RTSJ and imposing restrictions, for instance, on the nesting hierarchy of scopes.

Several works [2, 4, 13] have proposed an encoding of the nesting relation of scopes in the type system. The outlives relation between memory regions defined in [4] bears similarities to the relation $\succeq$ used in Sect. 3. One major difference is however that, unlike in [4], $\succeq$ represents only a snapshot of the nesting relation between scopes, thus allowing it to change during the program run.

5 Conclusion and Future Work

This paper presented a formalization of RTSJ's memory model which makes formal verification of real-time Java programs feasible. The described approach was implemented in the KeY system and successfully tested on several non-trivial examples. This does, however, not eliminate the need for SCJ profiles or programming guidelines constraining the use of scoped memory, since every sensible restriction imposed on it might ease its verifiability.

One direction of future work is to investigate how the low level relation $\preceq$ can be incorporated in more high-level specification means adding to less verbose specifications.

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(1):32–54, 2005.

[2] C. Andreae, Y. Coady, C. Gibbs, J. Noble, J. Vitek, and T. Zhao. Scoped types and aspects for real-time Java memory management. *Real-Time Syst.*, 37(1):1–44, 2007.

[3] G. Bollella and J. Gosling. The real-time specification for Java. *Computer*, 33(6):47–54, 2000.

[4] C. Boyapati, A. Salcianu, W. Beebee, and J. Rinard. Ownership types for safe region-based memory management in real-time Java. *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[5] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. *SIGPLAN Not.*, 34(10):1–19, 1999.

[6] D. Harel. Dynamic logic. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic*, volume 2 of *Extensions of Classical Logic*, pages 497–604. D. Reidel Publishing Company, 1984.

[7] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[8] J. Kwon, A. Wellings, and S. King. Ravenscar-Java: a high-integrity profile for real-time Java: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(5-6):681–713, 2005.

[9] J. Kwon and A. J. Wellings. Memory management based on method invocation in RTSJ. In *OTM Workshops*, pages 333–345, 2004.

[10] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

[11] M. Schoeberl, H. Sondergaard, B. Thomsen, and A. P. Ravn. A profile for safety critical Java. In *ISORC '07: Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 94–101. IEEE Computer Society, 2007.

[12] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 1997.

[13] T. Zhao, J. Noble, and J. Vitek. Scoped types for real-time Java. In *RTSS '04: Proceedings of the 25th IEEE International Real-Time Systems Symposium*, pages 241–251, Washington, DC, USA, 2004. IEEE Computer Society.