Procedure-Modular Specification and Verification of Temporal Safety Properties*

Siavash Soleimanifard[1], Dilian Gurov[1], and Marieke Huisman[2]

[1] KTH Royal Institute of Technology, Stockholm, Sweden
[2] University of Twente, Enschede, Netherlands

Abstract. This paper describes ProMoVer, a tool for fully automated procedure–modular verification of Java programs equipped with method–local and global assertions that specify safety properties of sequences of method invocations. Modularity at the procedure–level is a natural instantiation of the modular verification paradigm, where correctness of global properties is relativized on the local properties of the methods rather than on their implementations. Here it is based on the construction of maximal models for a program model that abstracts away from program data. This approach allows global properties to be verified in the presence of code evolution, multiple method implementations (as arising from software product lines), or even unknown method implementations (as in mobile code for open platforms).
ProMoVer automates a typical verification scenario for a previously developed tool set for compositional verification of control flow safety properties, and provides appropriate pre– and post–processing. Both linear–time temporal logic and finite automata are supported as formalisms for expressing local and global safety properties, allowing the user to choose a suitable format for the property at hand. Modularity is exploited by a mechanism for proof reuse that detects and minimizes the verification tasks resulting from changes in the code and the specifications. The verification task is relatively light–weight due to support for abstraction from private methods and automatic extraction of candidate specifications from method implementations. We evaluate the tool on a number of applications from the domains of Java Card and web–based application.

1 Introduction

In modern computing systems, code changes frequently. Modules (or components) evolve rapidly or exist in multiple versions customized for various users, and in mobile contexts, a system may even automatically reconfigure itself. As a result, systems are no longer developed as monolithic applications; instead they are composed of ready–made off–the–shelf components, and each component

* Soleimanifard's work is funded by the ContraST project of the Swedish Research Council VR, and Gurov's work by the EU FET project FP7–ICT–2009–3 HATS. Huisman's work is partially funded by ERC grant 258405 for the VerCors project.

may be dynamically replaced by a new one that provides improved or additional functionality. This static and dynamic *variability* makes it more important to provide formal correctness guarantees for the behaviour of such systems, but at the same time also more difficult. *Modularity* of verification is a key to providing such guarantees in the presence of variability.

In modular verification, correctness of the software components is specified and verified independently (*locally*) for each module, while correctness of the whole system is specified through a *global* property, the correctness of which is verified relative to the local specifications rather than relative to the actual implementations of the modules. It is this relativization that enables verification of global properties in the presence of static and dynamic variability. In particular, it allows an independent evolution of the implementations of individual modules, only requiring the re–establishment of their local correctness.

Hoare logic provides a popular framework for modular specification and verification of software, where it is natural to take the individual procedures as modules, in order to achieve scalability, see e.g., [22]. While Hoare logic allows the *local effect* of invoking a given procedure to be specified, temporal logic is better suited for capturing its *interaction with the environment*, such as the allowed sequences of procedure invocations. This paper shows that procedure–modular verification is also appropriate for control flow safety temporal logic: for each procedure the local property specifies its legal call sequences, while the system's global property specifies the allowed interactions of the system as a whole. Thus, temporal specifications provide a meaningful abstraction for procedures.

Control flow safety properties can be expressed in various formalisms, such as automata–based or process–algebraic notations, as well as in temporal logics such as LTL [30] or the safety fragment of the modal μ-calculus [19]. The approach that is described in this paper supports two of those formalisms, namely LTL and a variant of finite automata termed here safety automata. This is convenient in particular when writing properties of different nature and at different levels of abstraction and component granularity. Global specifications, for instance, are usually partial in nature, expressing certain critical requirements on the behaviour of the whole program. In contrast, local specifications should be as complete as possible, so that all interesting global properties are entailed. So, candidate local specifications extracted from an existing implementation would be more naturally represented with automata, while an abstract, global temporal restriction may be more naturally phrased in LTL. On the other hand, expressiveness of specification provides as usual convenience at the expense of algorithmic efficiency. Certain algorithmic problems, such as model checking and maximal flow graph construction (see below) are more efficiently solved if procedure calls are treated atomically, essentially reducing the context–free infinite–state behaviour of the program to its finite–state textual structure. The resulting restricted properties turn out to be adequate for specifying local properties of individual procedures (without self–calls), but are in general too inexpressive for higher levels of component granularity, and in particular for specifying global properties.

To support our approach, we have developed a fully automated verification tool, ProMoVer, which can be tried via a web–based interface [28]. It takes as input a Java program annotated with global and method–local correctness assertions written in temporal logic and it automatically invokes a number of tools from cvpp, a previously developed tool set for compositional verification [17], to perform the individual local and global correctness checks. Internally, cvpp uses the safety fragment of the modal μ-calculus as a property specification language, but ProMoVer also allows the user to write specifications in LTL, or as so–called safety automata, which are a variant of Schneider's security automata [27].

Essentially, ProMoVer is a wrapper that performs a standard verification scenario in the general tool set, to demonstrate that procedure–modular verification of temporal safety properties can be automated completely by using annotated programs as a single input. Importantly, ProMoVer only requires the public procedures to be annotated; the private ones are being considered merely as an implementation means. In addition, ProMoVer provides a facility to extract a method's legal call sequences by means of static analysis, given a concrete procedure implementation. A user thus does not have to write annotations explicitly; it suffices to inspect the extracted specifications and remove superfluous constraints that might hinder possible evolution of the code. Specifications can be extracted both in LTL and as safety automata, so a user can choose the formalism that is more appropriate for the problem at hand, or that he or she is most comfortable with. Finally, ProMoVer also practically supports modularity by providing proof storage and reuse: only the properties that are affected by a change (either in implementation or in specification) are reverified, all other results are reused.

We show validity of the approach on a number of Java programs from two application domains. Firstly, we perform experiments on some typical Java Card e-commerce applications. Such security–relevant applications are an important target for formal verification techniques. Here, we verify the absence of calls to non–atomic methods within transactions. Such properties, specifying legal call sequences for security–related methods, are an important class of platform–specific security properties. Secondly, we use an under–development web application to illustrate the verification of an open system in the presence of code evolution. Here, we verify that only a single connection to a database is created for each incoming request, and that it is properly closed. Properties of this type, specifying safe and efficient usage of a resource, are application–specific properties that are of major importance in the ICT business. The ProMoVer web interface allows the user to verify both platform– and application–specific properties, for which ready–made formalizations are provided.

To allow efficient algorithmic modular verification, the tool set currently abstracts away from all data, thus considering safety properties of the control flow; in particular, method calls in Java programs are over–approximated by non–deterministic choice on possible method implementations that the virtual call resolution might resolve to. This rather severe restriction on the *program model* facilitates the maximal model construction that is at the core of our modular

verification technique (see [13] for a proof of soundness and completeness for this program model). Still, many useful properties can be expressed at this level of abstraction. Besides the platform–specific and application–specific security properties discussed above, we can for example express properties such as: (*i*) a method that changes sensitive data is only called from within a dedicated authentication method, i.e., unauthorized access is not possible; or (*ii*) in a voting system, candidate selection has to be finished, before the vote can be confirmed. Extending the technique with data, either over finite domains or over pointer structures, will allow for a wider range of properties and possible applications, but requires a non–trivial generalization of the maximal model construction, and needs to be combined with abstraction techniques to control the complexity of verification and of model extraction from a program. We are currently investigating this.

The work in this paper is closely related to the development of CVPP [17]. As already pointed out, PROMOVER is essentially a wrapper that automates a typical verification scenario for CVPP, where modularity is applied at the procedure–level. In addition, PROMOVER provides support for different property specification languages, proof reuse, specification extraction, a collection of ready–formalized properties, and a translation between the different intermediate formats and formalisms. Results on a previous version of PROMOVER are reported in [29]. The present paper extends this earlier work by introducing an automata–based specification language and its modular verification principle. The use of the additional specification language is evaluated on a number of case studies, and is compared with the verifications based on the original LTL specifications. Furthermore, this paper presents an evaluation of PROMOVER on a significantly larger case study, representing an open system in the presence of code evolution.

Limitations. PROMOVER currently handles *procedure–modular verification* of control flow properties for sequential programs. The restriction to modularity at procedure level is meaningful (as we argue above) but not fundamental, and will be relaxed in future versions. As mentioned above, we are working on extending the method with data. The underlying theory for modeling multi–threaded programs has been developed earlier (see [16]), but the model checking problem is not decidable in general and has to be approximated suitably.

From a more practical point of view, the two main limitations are performance and the effort needed to write specifications. With respect to the first limitation, known theoretical bottlenecks are the maximal model construction and model checking of global properties (both exponential in the size of the formula), as well as the efficient extraction of precise program models (in particular concerning virtual call resolution and exception propagation). The support for proof reuse is our main means of addressing these bottlenecks. Notice also that the use of safety automata for specifying local properties eliminates the need to construct maximal models, since the automata themselves play the role of maximal models. As to the second limitation, to reduce the effort needed to write specifications, PROMOVER provides a library of common platform–specific

global properties, and a facility for extracting specifications from a given implementation, as explained above.

Related Work. A non–compositional verification method based on a program model closely related to ours is presented by Alur *et al.* [3]. It proposes a temporal logic CaRet for nested calls and returns (generalized to a logic for nested words in [1]) that can be used to specify regular properties of local paths within a procedure that skips over calls to other procedures. esp is another example of a successful system for non–compositional verification of temporal safety properties, applied to C programs [8]. It combines a number of scalable program analyses to achieve precise tracking (simulation) of a given property on multiple stateful values (such as file handles), identified through user–defined source code patterns. Maven is a modular verification tool addressing temporal properties of procedural languages, but in the context of aspects [11]. Recent work by Alur and Chauhuri proposes a unification of Hoare–style and Manna–Pnueli–style temporal reasoning for procedural programs, presenting proof rules for procedure–modular temporal reasoning [2].

Overview. The rest of this paper is organized as follows. Section 2 presents the use of ProMoVer from a user's point–of–view. Section 3 describes the underlying program model and Section 4 explains the property specification languages and compositional verification method based on constructing maximal models. Then, Section 5 describes the ProMoVer tool, while Section 6 describes several realistic case studies using the tool. Finally, the last section draws conclusions and suggests directions for future research.

2 ProMoVer: A User's View

We start by illustrating how ProMoVer is used on a small example. Both local method and global program properties are provided as assertions in the form of program annotations. We use a JML–like syntax for annotations (*cf.* [21]). ProMoVer is procedure–modular in the sense that correctness of the global program property is relativized on the local specifications of the individual methods. Thus, the overall verification task divides into two independent subtasks:

(*i*) a check that each method implementation satisfies its local specification, and
(*ii*) a check that the composition of local specifications entails the global property.

Notice that the second subtask only relies on the local specifications and does not require the implementations of the individual methods. Thus, changing a method implementation does not require the global property to be reverified, only the local specification. If the second subtask fails, ProMoVer translates the counterexample provided by the underlying tools into the form of a program behavior that is allowed by the local specifications, but violates the global one[3].

[3] Unfortunately, not all tools that we use provide counterexamples.

```
/**
 * @global_ltl_prop: even -> X ((even && !entry) W odd)
 */
public class EvenOdd {
   /** @local_interface: required odd
    *  @local_ltl_prop:
    *          G (X (!even || !entry) && (odd -> X G even))
    */
   public boolean even(int n) {
      if (n == 0)
        return true;
      else
        return odd(n-1);
   }

   /** @local_interface: required even
    *  @local_sa_prop:
    *          node s0 odd, entry
    *          node s1 odd, entry
    *          node s2 odd, entry, r
    *          edge s0 s0 tau
    *          edge s0 s1 odd caret even
    *          edge s0 s2 odd caret even
    *          edge s1 s1 tau
    *          edge s1 s2 tau
    */
   public boolean odd(int n) {
      if (n == 0)
        return false;
      else
        return even(n-1);
   }
}
```

Fig. 1: A simple annotated Java program

In addition to the properties, the technique also requires global and local *interfaces*. A global interface consists of a list of the methods *provided* (i.e., implemented) and *required* (i.e., used) by the program. The local interface of method m contains a list of the methods *required* by the method (as the provided method is obvious). ProMoVer can extract both global and local interfaces from method implementations.

Example 1. Consider the annotated Java program in Figure 1. It consists of two methods, `even` and `odd`. The program is annotated with a global control flow safety property expressed in LTL, and every method is annotated with a local property and an interface specifying the required methods. The local property of method `even` is expressed in LTL, while method `odd` is specified with a safety automaton. Here we only give an intuitive description of the properties specified in the example; formal definitions of the temporal logic LTL and safety automata are given in Section 4.

The global property expresses that "in every program execution starting in method `even`, the first call is not to method `even` itself". The local property of method `even` expresses that "method `even` can only call method `odd`, and after returning from the call, no other method can be called". The local property of

method `odd` is analogous but is expressed as a safety automaton (ASCII notation in Figure 1, and visualized in Figure 3 on page 12).

As mentioned above, the interfaces and local method specifications can be extracted from the method implementations automatically by ProMoVer (see Section 5).

As explained above, the annotated program is correct if (*i*) methods `even` and `odd` meet their respective local specifications, and (*ii*) the composition of all local specifications entails the global one. In fact, the annotated program is correct and our tool therefore returns an affirmative result.

Example 2. If we change the global property of the previous example to "in every program execution starting in method `even`, no call to method `odd` is made", the tool detects this and rechecks the global property for the already computed composition of local specifications. The local specifications do not have to be reverified. The verification of the global property fails. As a counterexample, ProMoVer returns the following program execution that is allowed by the local specifications, but violates the global one:

$$(\texttt{even}, \varepsilon) \xrightarrow{\texttt{even call odd}} (\texttt{odd}, \texttt{even}) \xrightarrow{\texttt{odd ret even}} (\texttt{even}, \varepsilon)$$

This counterexample, adapted for user understandability by replacing program points with the names of the methods they belong to (*cf.* Definition 4), should be understood as follows: from method `even`, method `odd` is called, and then method `odd` returns, and control is given back to `even`. This violates the desired global property, because `odd` is called from `even`.

3 Program Model

In this and the following section, we briefly present the formal framework underlying the ProMoVer tool that supports procedure–modular verification as illustrated above. It is heavily based on our earlier work on compositional verification [13, 12]. Here, we define our program model.

3.1 Models and Simulation

First, we formally define the abstract structure on which our program model and its operational semantics are based.

Definition 1 (Model). *A* model *is a (Kripke) structure* $\mathcal{M} = (S, L, \rightarrow, A, \lambda)$ *where* S *is a set of states,* L *a set of labels,* $\rightarrow \subseteq S \times L \times S$ *a labeled transition relation,* A *a set of atomic propositions, and* $\lambda : S \rightarrow \mathcal{P}(A)$ *a valuation, assigning to each state* s *the set of atomic propositions that hold in* s*. An* initialized model *is a pair* $(\mathcal{M}, E)$ *with* $\mathcal{M}$ *a model and* $E \subseteq S$ *a set of initial states.*

The definition of *simulation* on models is standard.

Definition 2 (Simulation). *A simulation on model $\mathcal{M}$ is a binary relation R on S such that whenever $(s,t) \in R$ then $\lambda(s) = \lambda(t)$, and whenever $s \xrightarrow{a} s'$ then there is some $t' \in S$ such that $t \xrightarrow{a} t'$ and $(s',t') \in R$. We say that t simulates s, written $s \leqslant t$, if there is a simulation R such that $(s,t) \in R$.*

Simulation on two models $\mathcal{M}_1$ and $\mathcal{M}_2$ is defined as simulation on their disjoint union $\mathcal{M}_1 \uplus \mathcal{M}_2$. The transitions of $\mathcal{M}_1 \uplus \mathcal{M}_2$ are defined by $in_i(s) \xrightarrow{a} in_i(s')$ if $s \xrightarrow{a} s'$ in $\mathcal{M}_i$ and its valuation by $\lambda(in_i(S)) = \lambda_i(S)$, where in_i (for $i \in \{1,2\}$) injects S_i into $S_1 \uplus S_2$. Simulation is extended to initialized models $(\mathcal{M}_1, E_1)$ by defining $(\mathcal{M}_1, E_1) \leqslant (\mathcal{M}_2, E_2)$ if there is a simulation R on $\mathcal{M}_1 \uplus \mathcal{M}_2$ such that for each $s \in E_1$ there is some $t \in E_2$ with $(in_1(s), in_2(t)) \in R$.

3.2 Flow Graphs

Our program model is based on the notion of *flow graph*, abstracting away from all data in the original program. It is essentially a collection of *method graphs*, one for each method of the program. Let *Meth* be a countably infinite set of methods names. A method graph is an instance of the general notion of initialized model.

Definition 3 (Method Graph). *A* method graph *for method $m \in Meth$ over a set $M \subseteq Meth$ of method names is an initialized model $(\mathcal{M}_m, E_m)$ where $\mathcal{M}_m = (V_m, L_m, \rightarrow_m, A_m, \lambda_m)$ is a finite model and $E_m \subseteq V_m$ is a non-empty set of* entry nodes *of m. V_m is the set of* control nodes *of m, $L_m = M \cup \{\varepsilon\}$, $A_m = \{m, r\}$, and $\lambda_m : V_m \to \mathcal{P}(A_m)$ so that $m \in \lambda_m(v)$ for all $v \in V_m$ (i.e., each node is tagged with its method name). The nodes $v \in V_m$ with $r \in \lambda_m(v)$ are* return points*.*

Notice that methods can have multiple entry nodes. Flow graphs that are extracted from program source have single entry points, but the maximal models that we generate for compositional verification may have several.

Every flow graph $\mathcal{G}$ is equipped with an *interface* $I = (I^+, I^-)$, denoted $\mathcal{G} : I$, where $I^+, I^- \subseteq Meth$ are the *provided* and externally *required* methods, respectively. These are needed to construct maximal flow graphs (see Section 4.2).

A flow graph is *closed* if its interface does not require any methods, and it is *open* otherwise. Flow graph *composition* is defined as the disjoint union $\uplus$ of their method graphs.

Example 3. Figure 2 shows the flow graph of the program from Figure 1. Its interface is $(\{\texttt{even}, \texttt{odd}\}, \emptyset)$, thus the flow graph is closed. It consists of two method graphs, for method `even` and method `odd`, respectively. Entry nodes are depicted as usual by incoming edges without source.

The operational semantics of flow graphs, referred to here as flow graph *behavior*, is also defined as an instance of an initialized model. We use transition label τ for internal transfer of control, $m_1 \,\mathsf{call}\, m_2$ for the invocation of method m_2 by method m_1 when method m_2 is provided by the program, $m_2 \;\mathsf{ret}\; m_1$ the corresponding return from the call, and label $m_1 \;\mathsf{caret}\; m_2$ for the (atomic) invocation of and return from an external method m_2 by method m_1.

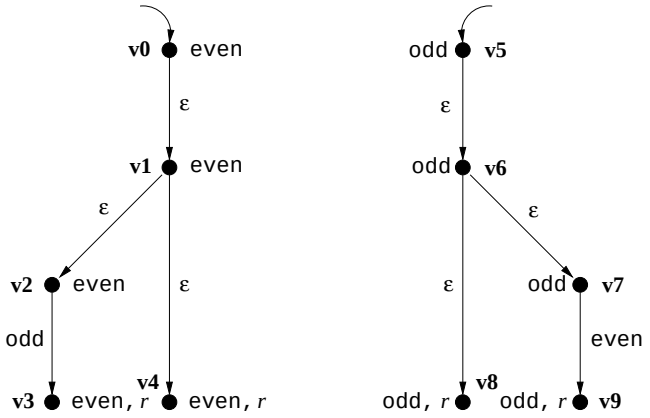


Fig. 2: Flow graph of EvenOdd

Definition 4 (Behavior). *Let $\mathcal{G} = (\mathcal{M}, E) : (I^+, I^-)$ be a flow graph such that $\mathcal{M} = (V, L, \to, A, \lambda)$. The behavior of $\mathcal{G}$ is defined as initialized model $b(\mathcal{G}) = (\mathcal{M}_b, E_b)$, where $\mathcal{M}_b = (S_b, L_b, \to_b, A_b, \lambda_b)$, such that $S_b = V \times V^*$, i.e., states (or configurations) are pairs of control points v and stacks σ, $L_b = \{m_1\ k\ m_2 \mid k \in \{\mathsf{call}, \mathsf{ret}\},\ m_1, m_2 \in I^+\} \cup \{m_1\ \mathsf{caret}\ m_2 \mid m_1 \in I^+ \wedge m_2 \in I^-\} \cup \{\tau\}$, $A_b = A$, $\lambda_b((v, \sigma)) = \lambda(v)$, and $\to_b \subseteq S_b \times L_b \times S_b$ is defined by the rules:*

$$\text{[transfer]}\ (v,\sigma) \xrightarrow{\tau} (v',\sigma) \quad \text{if } m \in I^+,\ v \xrightarrow{\varepsilon}_m v',\ v \models \neg r$$

$$\text{[call]}\ (v_1,\sigma) \xrightarrow{m_1\ \mathsf{call}\ m_2} (v_2, v_1' \cdot \sigma) \quad \text{if } m_1, m_2 \in I^+, v_1 \xrightarrow{m_2}_{m_1} v_1', v_1 \models \neg r, v_2 \models m_2, v_2 \in E$$

$$\text{[ret]}\ (v_2, v_1 \cdot \sigma) \xrightarrow{m_2\ \mathsf{ret}\ m_1} (v_1,\sigma) \quad \text{if } m_1, m_2 \in I^+, v_2 \models m_2 \wedge r, v_1 \models m_1$$

$$\text{[caret]}\ (v_1,\sigma) \xrightarrow{m_1\ \mathsf{caret}\ m_2} (v_1',\sigma) \quad \text{if } m_1 \in I^+, m_2 \in I^-, v_1, v_1 \xrightarrow{m_2}_{m_1} v_1', v_1' \models m_1, v_1 \models \neg r$$

The set of initial configurations is defined by $E_b = E \times \{\varepsilon\}$, where ε denotes the empty sequence over V.

Notice that return transitions always hand back control to the caller of the method. Calls to external methods are modeled with `caret` transitions that jump immediately from the external method invocation to the corresponding return, without considering the intermediate behavior. This treatment of method calls is inspired by the temporal logic CaRet [1] mentioned in the introduction, and is convenient for specifying the local behavior of flow graphs. When writing global specifications, however, one has to be aware that in this way possible callbacks from external methods are not captured.

Example 4. Consider the flow graph from Example 3. An example run through its (branching, infinite–state) behavior, from an initial to a final state, is:

$$(v_0,\varepsilon)\xrightarrow{\tau}(v_1,\varepsilon)\xrightarrow{\tau}(v_2,\varepsilon)\xrightarrow{\texttt{even call odd}}(v_5,v_3)\xrightarrow{\tau}(v_6,v_3)\xrightarrow{\tau}(v_8,v_3)\xrightarrow{\texttt{odd ret even}}(v_3,\varepsilon)$$

Now, consider just the method graph of method `even` as an open flow graph, having interface $(\{\texttt{even}\},\{\texttt{odd}\})$. The *local contribution* of method `even` to the above global behavior is the following run:

$$(v_0,\varepsilon)\xrightarrow{\tau}(v_1,\varepsilon)\xrightarrow{\tau}(v_2,\varepsilon)\xrightarrow{\texttt{even caret odd}}(v_3,\varepsilon)$$

Pushdown systems (PDS) and *Context Free Processes* (CFP) are alternative formalisms to express flow graph behavior (see e.g., [5]). We exploit this by using PDS model checking (concretely the tool MOPED [18]) and an own CFP model checker for verifying program behavior against temporal formulas [10].

4 Property Specification and Compositional Verification

In this section, we define the two main specification languages PROMOVER uses, namely *Linear–time Temporal Logic (LTL)* and *Safety Automata*, and introduce our compositional verification principles for both specification languages.

4.1 Property Specification

Safety properties can be expressed in a variety of formalisms. In this paper, we use two property specification languages: *safety LTL* which is the safety–fragment of *Linear–time Temporal Logic* (LTL) [23] that uses only the weak until–operator, and *Safety Automata* which are based on Schneider's *Security Automata* [27], but where states are additionally tagged with atomic propositions. Both specification languages demand a different treatment regarding verification. This subsection defines the syntax and semantics of the two specification languages, while the following one explains compositional verification for each case.

Linear–time Temporal Logic. One of the standard logics to express safety and liveness temporal properties is LTL. In our work, we focus on safety properties and therefore, we only use the safety fragment of LTL based on the weak version of until. The fragment is parameterized on a set of atomic propositions A as induced by a given flow graph $\mathcal{G}$, augmented with a special atomic proposition `entry` that holds at the entry nodes of $\mathcal{G}$.

Definition 5 (Safety LTL). *The formulae of* Safety LTL *are inductively defined by:*

$$\phi ::= p \mid \neg p \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \mathtt{X}\,\phi \mid \mathtt{G}\,\phi \mid \phi_1\,\mathtt{W}\,\phi_2$$

where p ranges over $A \cup \{\texttt{entry}\}$. For convenience, we sometimes use $p \Rightarrow \phi$ to abbreviate $\neg p \vee \phi$.

Satisfaction on states $(\mathcal{M}_b, s) \models \phi$ for LTL formulae is defined in the standard fashion [30]: formula $\texttt{X}\,\phi$ holds of state s in model $\mathcal{M}_b$ if ϕ holds in the next state of every run starting in s; $\texttt{G}\,\phi$ holds if for every run starting in s, ϕ holds in all states of the run; and $\phi\ \texttt{W}\ \psi$ holds in s if for every run starting in s, either ϕ holds in all states of the run, or ψ holds in some state and ϕ holds in all previous states.

Example 5. Consider the global property of class `EvenOdd` in Figure 1 (where `!`, `&&`, `||`, and `->` are ASCII notations for $\neg$, $\wedge$, $\vee$, and $\Rightarrow$, respectively) and its intuitive meaning discussed in Example 1. Flow graph extraction and construction ensure that entry nodes are only accessible via calls; hence, if control starts and remains in method `even`, execution can be at an entry node only as the result of a self–call. The formula thus states that "if program execution starts in method `even`, method `even` is not called until method `odd` is reached", which coincides with the interpretation given in Example 1.

Internally, the verification machinery for local LTL formulae is based on the safety fragment of the modal μ-calculus (that is, excluding diamond modalities and least fixed point recursion). Safety LTL is somewhat less expressive than the latter and can be uniformly encoded in it [7]. This translation is implemented as part of ProMoVer. As a technical detail, the additional atomic proposition `entry` that can appear in LTL formulae is removed during the translation.

Safety Automata. Alternatively, safety properties can be specified by means of safety automata, which are closely related to the notion of security automata [27].

Definition 6 (Safety Automaton). *A* safety automaton $\mathcal{A}$ *is an instance of an initialized model, where the set of labels is* $L_b = \{m_1\ k\ m_2 \mid k \in \{\mathsf{call}, \mathsf{ret}\}, m_1, m_2 \in I^+\} \cup \{m_1\ \mathsf{caret}\ m_2 \mid m_1 \in I^+ \wedge m_2 \in I^-\} \cup \{\tau\}$ *and the set of atomic propositions is* A.

Notice that since a safety automaton is an instance of the general notion of initialized model, the composition of two safety automata $\mathcal{A}_1$ and $\mathcal{A}_2$ is defined as their disjoint union $\mathcal{A}_1 \uplus \mathcal{A}_2$.

If a safety automaton $\mathcal{A}$ is used for specifying a method specification, then it can be translated in a straightforward manner into a flow graph $FG(\mathcal{A})$ that simulates exactly those flow graphs that are simulated by $\mathcal{A}$. Safety automaton $\mathcal{A}$ *simulates* a flow graph $\mathcal{G}$ if $\mathcal{G} \leqslant FG(\mathcal{A})$ as initialized models, as defined in Definition 2 (extended to initialized models).

The language of safety automata is equally expressive as μ-calculus and thus safety automata can be translated into μ-calculus formulae.

Example 6. Consider the local specification of method `odd` in Example 1, expressing "method `odd` can only call method `even`, and after returning from the

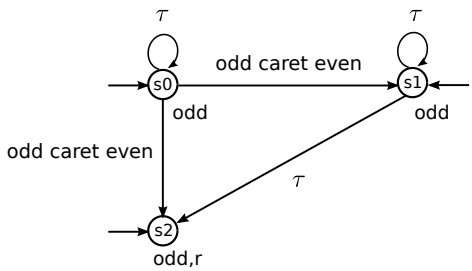


Fig. 3: Safety automaton for the local specification of method `odd`

call, no other method can be called". Figure 3 contains a graphical representation of this property.

The textual ASCII representation of the safety automaton is shown in Figure 1. In the ASCII representation, the `node` keyword defines a state of the automaton, followed by a list of comma–separated atomic propositions that hold in the state, while the `edge` keyword defines a transition of the automaton by starting state, target state, and the transition label, respectively. The atomic propositions `entry` and `ret` specify entry and return states, respectively, while label `tau` is the ASCII representation of τ.

Syntactic Sugar. Safety automata as defined above can become rather large in case of large interfaces. There are a variety of conventions one can use to facilitate a less verbose and more compact representation of an automaton. At present, we support negated labels to abbreviate that a particular action cannot be present on a transition between two states; for example, a label $\neg(a \ \mathsf{call} \ b)$ on a transition from an automaton state s_1 to state s_2 means that all labels from the label set L are present on the transition except for label $a \ \mathsf{call} \ b$. As another useful shorthand, it is often convenient to be able to express that the atomic proposition r may have any value in a particular state; for this we provide the "wild–card" atomic proposition $r*$.

Automata described with the above shorthands are easily translated into ordinary safety automata.

Example 7. The safety automaton from Figure 3 can be represented more compactly by the automaton illustrated in Figure 4. The latter automaton can be transformed (back) to the automaton of Figure 3 by duplicating state `s1` to states `s1` and `s2`, tagging only state `s2` with r, and eliminating all outgoing edges from state `s2`.

4.2 Compositional Verification

Next, we describe the compositional verification principles for the two specification languages. First, we describe compositional verification based on the construction of maximal flow graphs from the component's local specifications,

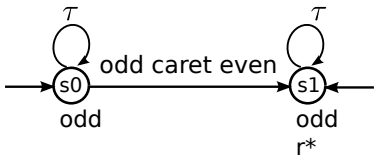


Fig. 4: Compact safety automaton for the local specification of method `odd`

when the latter are expressed in temporal logic: safety LTL, safety μ-calculus, or as modal equation systems (as defined by Larsen [20]). A modal equation system Σ is a finite set of defining equations of the shape $X = \phi_X$, where X is a propositional variable and ϕ_X is a formula of propositional modal logic without diamond modalities (recall that a modal formula $[l]\phi$ holds in a state s of a model if ϕ holds in all states accessible from s via transitions labeled with l). The defined variables X are pairwise distinct and bound in Σ, while all other variables are free. Its meaning is defined as its greatest solution. Modal equation systems are equivalent to the safety μ-calculus. In fact, we use this presentation of temporal properties in our maximal model construction and when automatically extracting local temporal specifications from method implementations (see Section 5).

The second part of this section discusses compositional verification when properties are expressed as safety automata.

Compositional Verification for Safety LTL. Our method for *algorithmic compositional verification* for LTL specifications is based on the construction of maximal flow graphs from component properties. For a given property ψ and interface I, consider the set of all flow graphs with interface I satisfying ψ. A *maximal flow graph* for ψ and I, denoted $\mathcal{M}ax(\psi, I)$, satisfies exactly those properties that hold for all members of the set. Thus, the maximal flow graph can be used as a representative of the set for the purpose of property verification. For details the reader is referred to [13].

For a system with k components, our principle of compositional verification based on maximal flow graphs can be presented as a proof rule with $k + 1$ premises.

$$\frac{\mathcal{G}_1 \models \psi_1 \;\; \cdots \;\; \mathcal{G}_k \models \psi_k \qquad \biguplus_{i=1,\ldots,k} \mathcal{M}ax(\psi_i, I_i) \models \phi}{\biguplus_{i=1,\ldots,k} \mathcal{G}_i \models \phi} \tag{1}$$

The rule states that the composition of components $\mathcal{G}_1 : I_1, ..., \mathcal{G}_k : I_k$ satisfies a global property ϕ if there are local properties ψ_i such that (*i*) each component $\mathcal{G}_i$ satisfies its local property ψ_i, and (*ii*) the composition of the k maximal flow graphs $\mathcal{M}ax(\psi_i, I_i)$ satisfies ϕ. This principle is proved *sound* and *complete* in [13].

In the context of ProMoVer, we consider individual program methods as components. If we instantiate the above compositional verification principle to procedure–modular verification, we obtain the verification tasks stated informally in Section 2 (where M is the set of program methods, with $k = |M|$, and ψ_i and $\mathcal{C}_i$ are the specification and the implementation of method m_i, respectively):

(i) **Checking $\mathcal{C}_i \models \psi_i$ for $i = 1, ..., k$:** For each method $m_i \in M$, (a) extract the method graph $\mathcal{G}_i$ from $\mathcal{C}_i$, and (b) model check $\mathcal{G}_i$ against ψ_i. For the latter, we exploit the fact that flow graphs are *Kripke structures*, and apply standard finite–state model checking.

(ii) **Checking $\biguplus_{i=1,...,k} \mathcal{M}ax(\psi_i, I_i) \models \phi$:** ($a$) Construct maximal flow graphs $\mathcal{M}ax(\psi_i, I_i)$ for all method specifications ψ_i and interfaces I_i, then (b) compose the graphs, resulting in flow graph $\mathcal{G}_{\mathcal{M}ax}$, and finally ($c$) model check $\mathcal{G}_{\mathcal{M}ax}$ against global property ϕ. For the latter, represent the behavior of $\mathcal{G}_{\mathcal{M}ax}$ as a PDS and use a standard PDS model checker.

Compositional Verification for Safety Automata. When all specifications are specified by safety automata, we check (i) whether the safety automaton of each method simulates its method graph, and (ii) whether the composition of the flow graphs of all local automata is simulated by the global automaton. Notice that in (ii) the flow graphs of the local safety automata serve as "maximal" flow graphs. This is due to that fact that, by definition, the safety automaton specification of a method simulates exactly those method graphs that satisfy the specification. Thus, the general compositional verification principle in this case for a system with k methods can be presented as the following proof rule.

$$\frac{\mathcal{G}_1 \leqslant \mathcal{A}_1 \ \cdots \ \mathcal{G}_k \leqslant \mathcal{A}_k \qquad \biguplus_{i=1,...,k} FG(\mathcal{A}_i) \leqslant \mathcal{A}}{\biguplus_{i=1,...,k} \mathcal{G}_i \leqslant \mathcal{A}} \tag{2}$$

The principle states that the composition of method graphs $\mathcal{G}_1 : I_1, ..., \mathcal{G}_k : I_k$ satisfies a global property expressed by a safety automaton $\mathcal{A}$ if there are local properties expressed by safety automata $\mathcal{A}_i$ such that (i) each method graph $\mathcal{G}_i$ is simulated by its local property $\mathcal{A}_i$, and (ii) the composition of the k flow graphs of the local safety automata $\mathcal{A}_i$ is simulated by $\mathcal{A}$. *Soundness* and *completeness* of this principle is established similarly as soundness and completeness of Principle 1 (in [13]).

In ProMoVer, for safety automata specifications, the verification tasks stated informally in Section 2, are achieved based on Principle 2 by:

(i) **Checking $\mathcal{C}_i \leqslant \mathcal{A}_i$ for $i = 1, ..., k$:** For each method $m_i \in M$, (a) extract the method graph $\mathcal{G}_i$ from $\mathcal{C}_i$, and (b) check that $\mathcal{G}_i$ is simulated by $\mathcal{A}_i$. For the latter, we exploit the fact that flow graphs and safety automata are initialized models, and check for simulation accordingly.

(ii) **Checking $\biguplus_{i=1,\ldots,k} FG(\mathcal{A}_i) \leqslant \mathcal{A}$:** ($a$) compose the flow graphs of the safety automata specifications of all methods, resulting in safety automaton FG_{comp}, and then (b) model check FG_{comp} against global automaton $\mathcal{A}$. For the latter, represent the behavior of FG_{comp} as a context free process, and use a CFP model checker (on the temporal formula translation of the automaton).

The two principles can be combined freely, so that local specifications and global properties can be written in either formalism. In task (i), if method m is specified in LTL, the flow graph extracted from method m is model checked against the specification, while if method m is specified with a safety automaton, simulation of the flow graph by the safety automaton is checked instead. In task (ii), maximal flow graphs are constructed for all methods with LTL specifications, and are then composed with the flow graphs of all safety automata specifications. Finally, if the global property is specified in LTL, the composition result is model checked against the property, while if the global property is specified by a safety automaton, the composition result is model checked against the automaton instead.

Example 8. Consider again the annotated Java program from Example 1. In the example, the global property and the local specification of method `even` are specified in LTL, while the local specification of method `odd` is given as a safety automaton. ProMoVer first extracts the method graphs of methods `even` and `odd`, denoted $\mathcal{G}_{even}$ and $\mathcal{G}_{odd}$, respectively. Next, ProMoVer checks $\mathcal{G}_{even} \models \psi_{even}$ and $\mathcal{G}_{odd} \leqslant \mathcal{A}_{odd}$. Independently, it constructs the maximal flow graph of method `even` denoted $\mathcal{M}ax(\psi_{even}, I_{even})$ and composes it with the flow graph of the safety automaton of method `odd` denoted FG_{odd} to obtain the flow graph $FG_{even-odd} = \mathcal{M}ax(\psi_{even}, I_{even}) \uplus FG_{odd}$. Finally, ProMoVer translates $FG_{even-odd}$ to a PDS and model checks the latter against the global LTL property.

5 The ProMoVer Tool

Next we describe the internals of ProMoVer. As mentioned above, ProMoVer essentially is a wrapper for cvpp [17], with extra features such as specification extraction, private method abstraction, a property specification library and support for proof reuse. All features are implemented in Python. ProMoVer can be tested via a web interface [28].

CVPP Wrapper. Figure 5 shows schematically how ProMoVer combines the individual cvpp tools. An annotated Java program, as exemplified in Section 2, is given as input. The *pre–processor* parses the annotations, using the Java Doclet API [9], and then passes properties and interfaces on to the different cvpp tools.

Task (i) first invokes the Analyzer tool described in [4] to extract the method graphs of the program. This tool builds on Sawja [15] to extract flow graphs from Java bytecode. Then our Graph tool is used. This implements several algorithms on flow graphs and safety automata, including composition $\uplus$

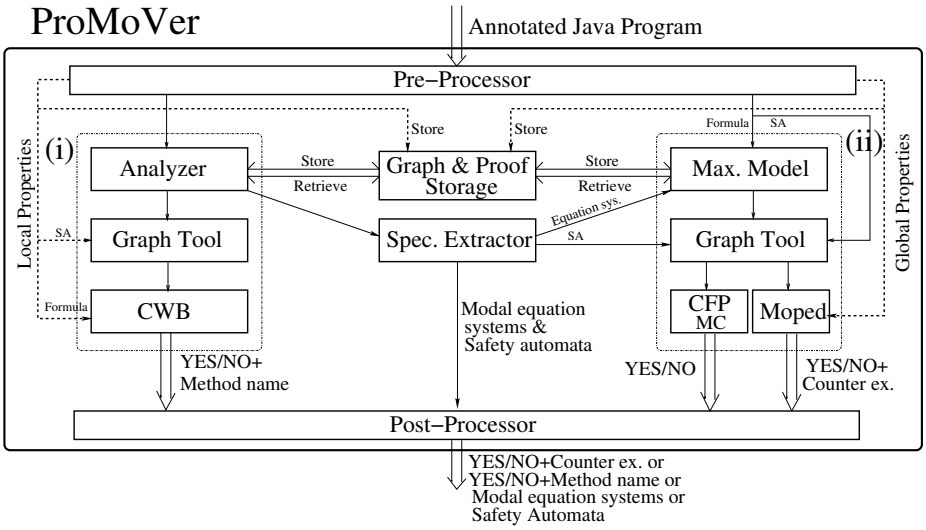


Fig. 5: Overview of ProMoVer and its underlying tool set

and translations of flow graphs and safety automata into different formats. Here the Graph tool is used to translate the flow graph of each method into a CCS model. These are then checked against the respective local method specifications using the *Concurrency Workbench* (cwb) [6]. If the specification is specified by LTL then it is translated to a μ-calculus formula and cwb is used to model check the CCS model against the formula. In case the specification is given in safety automaton, it is also translated into a CCS model and language inclusion is checked by cwb.

Task (*ii*) first constructs a maximal flow graph for every method specified with LTL by using the Maximal Model tool, and for methods specified with safety automata translates the specifications to flow graphs by the Graph tool. Then the Graph tool composes the generated flow graphs and converts the result into a PDS (for a global property expressed with LTL) or CFP (for a global property expressed as a safety automaton). Finally Moped [18] is used to model check the PDS against the LTL global property or CFP MC [10] is used to model check the CFP against the μ-calculus translation of the global safety automaton. The latter is a model checker implemented as part of the toolset.

The *post–processor* collects all model checking results and converts these into a user–understandable format. It only returns a positive result if all collected model checking tasks succeed. If one of the local model checking tasks fails, the name of the method that violates its specification is returned. If the global model checking task fails, for LTL global properties, a counterexample is provided by Moped and translated into a program execution and returned, however, for safety automata global properties, CFP MC does not provide a counterexample and therefore, no counterexample is returned.

Specification Extraction. To reduce the effort needed to write specifications, ProMoVer provides support to extract a specification from a given method implementation, resulting in the (over–approximated) order of method invocations for this method. The user might then want to remove some superfluous dependencies, in order not to be overly restrictive on possible evolution of the code.

ProMoVer extracts specifications in two different formats: modal equation systems and safety automata. Modal equation systems have the advantage that in cvpp they can serve directly as input for the construction of maximal flow graphs. On the other hand, the extracted safety automata specifications bypass the expensive maximal flow graph construction process, are often more intuitive, and can be modified graphically.

Consider again Figure 1. Specification extraction for method `odd` results in the following modal equation system (where `eps` is ASCII notation for ε, and `ff` denotes *false*):

```
@local_eq_prop: (X0){ X0  =  [even]X1 /\ [odd]ff  /\ [eps]X0;
                      X1  =  [odd]ff  /\ [even]ff /\ [eps]X1; }
```

The formula (which refers to the denotation of `X0` in the greatest solution of the equation system) essentially specifies that method `even` may be called at most once: initially `X0` holds, and method `even` may be called or an internal step (labeled `eps`) may be made. After calling `even`, `X1` should hold and only internal steps are allowed.

Using the specification extractor to extract the safety automaton specification for the same method results in the safety automaton depicted in Figure 3.

As a more involved example, consider the following method `m` together with its specification, extracted as a modal equation system:

```
@local_eq_prop:
(X0){ X0  =  [m4]ff /\ [m1]X1 /\ [m3]ff /\ [m2]ff /\ [m]ff /\ [eps]X0;
      X1  =  [m4]ff /\ [m1]ff /\ [m3]ff /\ [m2]X2 /\ [m]ff /\ [eps]X1;
      X2  =  [m4]X3 /\ [m1]ff /\ [m3]X4 /\ [m2]ff /\ [m]ff /\ [eps]X2;
      X3  =  [m4]ff /\ [m1]ff /\ [m3]ff /\ [m2]ff /\ [m]ff /\ [eps]X3;
      X4  =  [m4]ff /\ [m1]ff /\ [m3]ff /\ [m2]ff /\ [m]ff /\ [eps]X4;
    }

public void m() { int i = m1(); int j = m2();
                  if (i < j) { m3(); } else { m4(); } }
```

The formula captures that first only `m1` can be called, then only `m2`, and then either `m3` or `m4`, and no further calls can be made. Suppose that the order of invoking `m1` and `m2` is immaterial for this program. In that case a designer may choose to change the equations defining `X0` and `X1` to allow the two methods to be called in any order (whereas the defining equations for `X2` to `X4` remain unchanged):

```
X0  =  [m4]ff /\ [m1]X10 /\ [m3]ff /\ [m2]X11 /\ [m]ff /\ [eps]X0;
X10 =  [m4]ff /\ [m1]ff  /\ [m3]ff /\ [m2]X2  /\ [m]ff /\ [eps]X10;
X11 =  [m4]ff /\ [m1]X2  /\ [m3]ff /\ [m2]ff  /\ [m]ff /\ [eps]X11;
```

Using the specification extractor to extract the safety automaton specification of method `m` above will result in the following safety automaton, illustrated graphically in Figure 6.

```
node s1 m,entry      edge s1 s1 tau      edge s1 s2 m caret m1
node s2 m            edge s2 s2 tau      edge s2 s3 m caret m2
node s3 m            edge s3 s3 tau      edge s3 s4 m caret m3
node s4 m,r*         edge s4 s4 tau      edge s3 s5 m caret m4
node s5 m,r*         edge s5 s5 tau
```

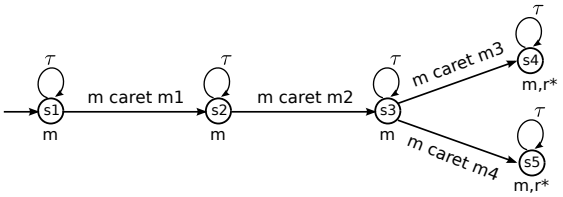


Fig. 6: Extracted safety automaton

As above, also the safety automaton can be relaxed for the case that the order in which the methods `m1` and `m2` are invoked is immaterial, as shown in Figure 7.

```
node s1 m,entry      edge s1 s1 tau      edge s1 s2_1 m caret m1
node s2_1 m          edge s2_1 s2_1 tau  edge s2_1 s3 m caret m2
node s2_2 m          edge s2_2 s2_2 tau  edge s1 s2_2 m caret m2
node s3 m            edge s3 s3 tau      edge s2_2 s3 m caret m1
node s4 m,r*         edge s4 s4 tau      edge s3 s4 m caret m3
node s5 m,r*         edge s5 s5 tau      edge s3 s5 m caret m4
```

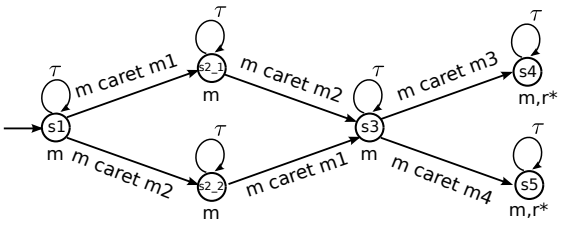


Fig. 7: Relaxed safety automaton

Private Method Abstraction. Since private methods are used as a means of implementation for public methods, at the flow graph level, all calls to private methods can be inlined into the flow graph of the public methods. The resulting method flow graphs thus only describe the public behavior, and users only have to specify the public methods. For details the reader is referred to [13].

Property Specification Library. ProMoVer's web interface provides a collection of pre–formalized global properties. These describe platform–specific security properties, restricting calls to API methods. Currently, the library contains several Java Card and voting system properties.

Proof Storage and Reuse. All extracted method flow graphs and constructed maximal flow graphs are stored when a program is verified by ProMoVer. If later the implementation of method m changes, a new method flow graph is extracted and checked against m's local specification. If m's local specification ϕ_m changes, the existing flow graph of method m is model checked against ϕ_m. In addition a new maximal flow graph for m is constructed from ϕ_m. This is composed with the other maximal flow graphs (recovered from storage), and the composed flow graph is model checked against the global property.

6 Experimental Results for ProMoVer

We use ProMoVer to verify standard control flow safety properties on a number of applications from two application domains where code evolution is important, namely Java Card and web-based applications.

6.1 Experiments on Java Card Applications

Java Card is one of the leading interoperable platforms for smart cards. Many smart card applications are security–critical. As mentioned above, for platforms such as Java Card, collections of control flow safety properties exist that programs should adhere to in order to provide minimal security requirements. We focus on such a property of the Java Card transaction mechanism. This mechanism ensures that data remains consistent upon power loss, however, careful use of it sometimes demands that certain methods are not used within a transaction. We show how this global safety property can be expressed in our setting, and be verified with ProMoVer for several applications, where we apply specification extraction to annotate the public methods of the applications.

As a side remark, control flow of Java Card programs might be different from control flow of a standard Java application, for example the Java Card firewall can cause an object field to raise an exception. Handling these differences correctly is an issue for the control flow graph extraction algorithm. However, for the properties and case study discussed here, this difference in control flow is not relevant, and we do not discuss it further here.

Application	#LoC	#Methods (Public)	#Calls (Relevant)
`AccountAccessor`	190	9 (7)	38 (4)
`TransitApplet`	918	18 (5)	106 (5)
`JavaPurse`	884	19 (9)	190 (25)

Table 1: Applications details

The Java Card Transaction Mechanism. Smart cards have two types of writable memory, *persistent memory* (EEPROM or Flash) and *transient memory* (RAM). Transient memory needs constant power supply to store information, while persistent memory can store data without power. Smart cards do not have their own power supply; they depend on the external source that comes from the card reader device. Therefore, a problem known as *card tear* may occur: a power loss when the card is suddenly disconnected from the card reader. If a card tear occurs in the middle of updating data from transient to persistent memory, the data stored in transient memory is lost and may cause the smart card to be in an inconsistent state.

To prevent this, the *transaction mechanism* is provided. It can be used to ensure that several updates are executed as a single *atomic* operation, *i.e.*, either all updates are performed or none. The mechanism is provided through methods `beginTransaction` for beginning a transaction, `commitTransaction` for ending a transaction with performed updates, and `abortTransaction` for ending a transaction with discarded updates [14] – all declared in class `JCSystem` of the Java Card API.

However, the Java Card API also contains some *non–atomic* methods that are better not used when a transaction is in progress. Notably, the class `javacard.framework.Util` that provides functionality to store and update byte arrays, contains methods `arrayCopyNonAtomic` and `arrayFillNonAtomic`. Careful use of the transaction mechanism can require that these methods should not be used within a transaction. We use ProMoVer to verify that applications comply with this *Transaction Policy*.

The Applications. For this experiment we use several public examples of Java Card applications. All are realistic e-commerce applications developed by Sun Microsystems to demonstrate the use of the Java Card environment for developing e-commerce applications. `AccountAccessor` is an application to keep track of account information. It is to be used by a wireless device connected via a network service. It contains methods to look up and modify the account balance. `TransitApplet` implements the on–card part of a system that connects to an authenticated terminal and provides account information and operations to modify the account balance. `JavaPurse` is a smart card electronic purse application providing secure money transfers. It contains a balance record denoting the user's current and maximum credits, and methods to initialize, perform and complete a secure transaction. Further, it also contains methods to update in-

formation related to a loyalty program, and to validate and update the values of transactions, balance and PIN code.

Table 1 shows information about the size, number of methods (total and public), and number of method invocations (total and relevant for the global property) of these applications.

Specification of the Transaction Policy. As discussed above, we want to ensure formally that the non-atomic methods `arrayCopyNonAtomic` and `arrayFillNonAtomic` are not invoked within a transaction. Hence, applications have to adhere to the following global control flow safety property:

> In every program execution, after a transaction begins, methods `arrayCopyNonAtomic` and `arrayFillNonAtomic` are not called until the transaction ends.

This safety property can be expressed formally with the following LTL formula:

$$\mathtt{G}\,(\mathtt{beginTransaction} \Rightarrow ((\neg\mathtt{arrayCopyNonAtomic} \wedge \neg\mathtt{arrayFillNonAtomic})\,\mathtt{W}\,\mathtt{commitTransaction}))$$

The property could also have been specified, though more verbosely, as a safety automaton.

Local Method Specifications. In order to compare the efficiency of verification for the different formalisms for writing local specifications, we annotated the methods of each application once in LTL and once with safety automata. For this we used the assistance of the specification extraction facility of ProMoVer.

The specification extractor is used to obtain local specifications for every public method, either as an equation system or as a safety automaton. The extracted specifications describe the actual order of method invocations in the code. We then inspect the specifications for immaterial orderings and remove these, with the intention that local method specifications should only restrict unwanted sequences of method calls made from within the specified method.

Writing specifications abstractly allows for possible evolution of the method implementations. Comparing the two formalisms, it can be observed that using temporal logic allows in general for more compact specifications, since only explicitly prohibited method invocations have to be mentioned.

Verification Results. After annotating the applications with global properties and local specifications, ProMoVer extracts the flow graphs of the applications and partitions these into the individual method graphs to verify adherence to the local specifications. Further, for applications with local specifications given in LTL, the maximal method graphs are constructed from the specifications, and their composition is verified *w.r.t.* the global property above. For applications with local specifications given as safety automata, the corresponding flow graphs of the automata are composed and verified *w.r.t.* the global property.

The statistics for these verifications are summarized in Table 2 and 3. The tables show: the time spent by the pre–processor (PPT) and the graph extractor

Application	PPT	GE	#NEF	LMC	MFC	#NMF	GMC	TT
`AccountAccessor`	1.4	3.8	435	0.5	0.7	20	0.9	8.7
`TransitApplet`	1.4	4.7	897	0.5	0.9	30	0.9	13.2
`JavaPurse`	1.5	6.5	1543	0.5	13.0	48	1.1	22.5

Table 2: Verification Results with LTL Local Specifications

Application	PPT	GE	#NEF	LMC	GMC	TT
`AccountAccessor`	1.4	3.8	435	0.6	0.9	8.1
`TransitApplet`	1.4	4.7	897	4.0	0.9	12.2
`JavaPurse`	1.5	6.5	1543	4.8	1.0	14.8

Table 3: Verification Results with Safety Automata Local Specifications

(GE) (all times here and below are in seconds), the number of nodes in the extracted flow graphs (#NEF), the time spent for local model checking (LMC) and for constructing maximal flow graphs (MFC), the number of nodes in the maximal flow graph composition (#NMF), the time spent for global model checking (GMC), and the total time spent for the whole verification task including conversions between formats and post–processing (TT). All results are obtained on a SUN SPARC machine. Notice that the pre–processing time (PPT), the graph extraction time (GE), and the number of nodes in the extracted flow graphs (#NEF) are the same for applications with local specifications given in LTL and safety automata, but in the case of safety automata the expensive process of maximal flow graph construction is bypassed.

As can be observed from the tables, local model checking takes longer for applications with local specifications given as safety automata. This is due to the higher verbosity of local specifications with automata, compared with temporal logic formulae, as discussed above. However, the increased local model checking time is compensated for by the translation from automata into method graphs, which just renames transition labels and is thus much less expensive than the corresponding maximal model construction for temporal logic specifications.

Proof Reuse. We also evaluate experimentally the advantages of exploiting the proof storage and reuse mechanism. After the first verification, when all method and maximal flow graphs have been stored, we changed, for each application, once the source code and once the local specification of a public method, and used ProMoVer to re–verify the applications.

The changes in the source code imitate a typical code evolution scenario, where a method's body is changed, for example, for the purpose of maintenance. The changes in the local specifications are motivated by the scenario where the (automatically extracted) specifications are weakened to support code evolution.

The results of proof reuse are shown in Table 4. The table shows: maximal flow graph construction time (MFC), the time spent by ProMoVer to re–verify

	Code Change		Local Specification Change		
Application	New TT	% TT	MFC	New TT	% TT
`AccountAccessor`	6.0	68	0.1	4.6	52
`TransitApplet`	7.2	54	0.1	5.0	37
`JavaPurse`	9.0	40	0.1	5.4	24

Table 4: Proof Reuse Results

the program after the change (new TT), and its percentage in relation with the original verification time (%TT). The numbers indicate that proof reuse can significantly reduce the verification time, especially for larger applications.

6.2 Experiments on a Web Application

Web applications are client–server programs intended to be used over the Internet. Typically, clients are *web browsers* and servers are *web servers*. Such web applications are of major importance in the ICT business, and therefore it is crucial to check that they function correctly, without any unexpected errors.

To minimize errors, various coding standards exist that components of web applications should respect. Based on these standards, we identify several requirements for database connections and transactions of the *Java Enterprise* platform that can be expressed as control flow safety properties. We show how ProMoVer is used to verify such control flow database connection properties in the presence of code evolution. Concretely, we verify the *Single DataBase Connection Policy* for an incomplete and prototype version of the Sail–Web application (both property and application are discussed in more detail below). First, we verify the incomplete program with the specifications of the missing components. Later, we import the missing code from the prototype into the incomplete code and re–verify the program. By this we mimic the code evolution scenario discussed above and how it is supported by ProMoVer.

Java Enterprise Platform (J2EE). J2EE is a popular platform to develop Java web applications. It provides an API and specification of the runtime environment to develop and run typical enterprise applications. In J2EE, a web application consists of a set of components running on a web server. These components are typically used by the web server to extend its capabilities for generating responses to clients' requests.

A commonly used technology to develop such components is *Java Servlets*. Technically, servlets are Java classes that conform to the Java–Servlet API model. They may be used by developers to provide web–pages containing dynamic contents (e.g., HTML or XML) using the Java platform. Servlets are typically invoked via the methods `doPost` and `doGet`. The web server creates instances of the servlets at boot time and maintains these objects throughout the execution. When a request arrives from a client, the web server assigns a thread from a thread–pool to the request and forwards the request to the `doPost` or

`doGet` methods of the suitable servlet. The servlet computes a response for the request and returns it back to the web server. Then, this response is sent back to the client and the allocated thread is returned back to the thread–pool.

Web servers use multi–threading to be able to respond to simultaneous requests; however, each request is handled by a single thread. Hence, control flow properties for processing a single request can be analyzed in a non–concurrent setting.

J2EE Database Connection. Web applications often use databases to manipulate data and store information. For example, almost all web applications that provide support for user accounts store user information (such as user name and password) in a database.

Typical examples of control flow properties for database connections are the *safe database transaction policy* that states that "a database transaction should be either committed or rolled–back if an exception is raised", and the *database connection policy* that states that "only a single database connection should be created for each request and it should be properly closed". In the remainder of this section, we focus on the second property. The first property can be expressed and verified similarly to the Java Card transaction policy presented above, and therefore we do not discuss its verification here.

To understand why the *database connection policy* is important, one should realize that each database system is capable of handling a limited number of simultaneous connections only. Therefore, if a single request opens more than one connection to a database, it is using these limited resources inefficiently. Moreover, such a practice significantly increases the likelihood of coding–errors caused by not closing the open connections properly. Therefore, the database connection policy demands that web applications obtain only a single database connection per request, and moreover, that this connection is closed before the assigned thread is returned back to the pool.

Various strategies and frameworks exist that ensure that the policy is respected, such as using filters or frameworks like JBoss Seam and Spring. However, many web programmers do not use any of these facilities. Therefore, it is highly desirable to have a tool that can check such properties of web applications.

Formal Specification of the Single Database Connection Policy. If no special framework is used, Java applications typically communicate with a database via the Java DataBase Connectivity (JDBC) API. In this API, the methods `java.sql.DriverManager.getConnection` and `java.sql.Connection.close` are used to create and close database connections, respectively. Therefore, in order to check the database connection property explained above, we check the absence of consecutive calls to the method `java.sql.DriverManager.getConnection` unless the method `java.sql.Connection.close` is called in between.

More precisely, this means that applications should respect the following global control flow safety property:

Sail-Web App.	#LoC	#Classes (Servlets)	#Public Methods
Limited Package	3038	20 (16)	28
Extended Package	10844	32 (28)	94

Table 5: Sail-Web application details

> In every thread execution, after a connection to a database is created, the method `java.sql.DriverManager.getConnection` is not called until the connection is closed.

This safety property can be formally expressed by the following safety LTL formula:

```
G (p.DriverManager.getConnection ⇒
    X (¬p.DriverManager.getConnection W p.Connection.close))
```

where `p` abbreviates the `java.sql` package.

The Sail–Web Application. For our experiments, we use the Sail–Web (Scalable Architecture for Interactive Learning on the Web) application, which is available in Google Codes [25]. Sail–Web is an ongoing project that aims at developing a web–based content management system for interactive learning. This application uses a MySql database through the JDBC API to manipulate data. The application is divided into two separate packages, here called *limited* and *complete*. The complete package is an extended version of the limited one, supporting several additional features.

Table 5 shows information about size, number of classes (total and servlets), and number of public methods of the limited package and its extension with some features imported from the complete one. The extended package includes 12 more classes, here called *additional classes*. These classes extend the limited package by adding new features such as file management, URL connection, and security utilities. We begin our verification experiment with the code of the limited package, with additional annotations specifying the control flow of the methods of additional classes. This resembles systems with unavailable code, e.g., mobile code. Then, to imitate the code evolution scenario, we import the code of the additional classes into the limited package (which forms extended package) and re–verify the program.

Focusing on the database connection policy, private methods `createConnection` and `shutdown` of the servlets are used to create and close database connections, respectively. The code of these methods is shown in Figure 8.

These two methods are invoked by the `doGet` and `doPost` methods of servlets. As an example, the code of method `doGet` of class `VLEGetAnnotations` is shown in Figure 9. Methods `doGet` and `doPost` of other servlets use similar code to respond to the requests. Method `getData` is a private method to process requests; it has a different implementation in each servlet.

```
private static void createConnection() {
  try {
    //create a connection to the mysql db
    Class.forName("com.mysql.jdbc.Driver").newInstance();
    conn = DriverManager.getConnection(DBurl, "sailuser", "sailpass");
  } catch (Exception ex) { ex.printStackTrace(); }
}

private static void shutdown() {
  try {
    conn.close();
  } catch(SQLException ex) { ex.printStackTrace(); }
}
```

Fig. 8: The private methods to create and close database connections

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
                  throws ServletException, IOException {
  createConnection();
  getData(request, response);
  shutdown();
}
```

Fig. 9: The code of method `doGet` of `VLEGetAnnotations` class

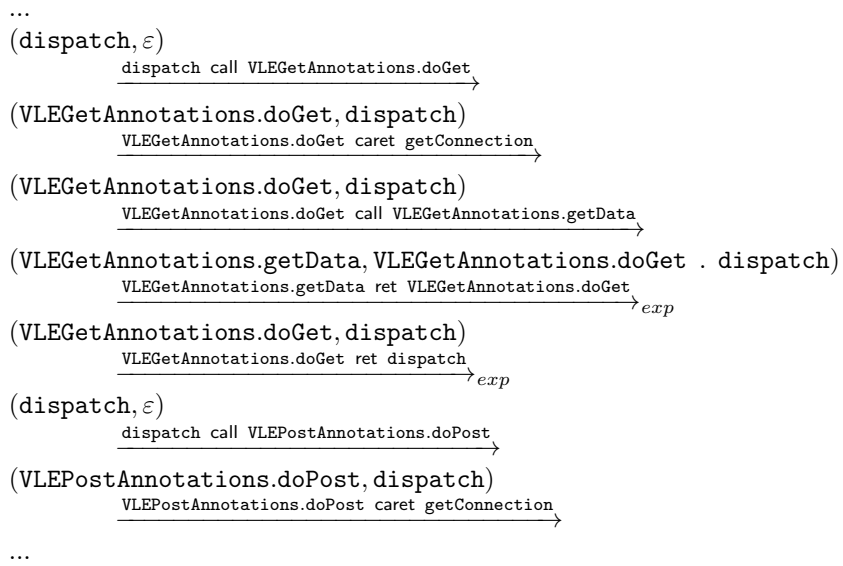
As explained above, the web server invokes the objects of the servlets based on the input request. We have modeled the behaviour of the web server by implementing a method that iteratively forwards random requests to random Servlets in a loop. This method is called `dispatch`.

Verification Results. We used the specification extractor to extract safety automata specifications of the methods of the Sail–Web application. The extracted safety automata represent the actual order of method invocations in the program. As mentioned above, we also annotated the specifications of the methods of the additional classes into the application and used these for verification of the global safety control flow property expressing the database single connection policy. ProMoVer constructs maximal models of the annotated specifications, combines these with the extracted specifications into a PDS and model checks the result against the global property. The statistics for the verification are given in the first row of Table 6. In the table, we show the time spent by the pre–processor (PPT), graph extractor (GE), local model checking (LMC), maximal flow graph construction (MFC), global model checking (GMC), and the whole verification (TT). Notice that in this version of the program local model checking is not used because the local specifications are extracted from the code and need not be checked. The verification result is "NO" and the following counterexample execution in the form of a program behaviour is returned[4].

[4] To simplify the presentation, the package names are removed from the configurations.

Sail-Web App.	PPT	CG	LMC	MFC	GMC	TT
Limited Package	43	19	–	8	1	71
Limited Package (with improvements)	2	19	–	–	1	22
Extended Package	–	–	32	–	–	32

Table 6: Verification results of the Sail-Web application



where the exceptional transitions are labeled by *exp*.

The counterexample shows an execution starting in method `dispatch` that results in two simultaneous connections to the database. The reason is that after creating the first connection, if an unhandled runtime exception (e.g., NullPointerException) is raised in method `getData` of class `VLEGetAnnotations`, then the normal execution path of the program changes. In the counterexample, the first unhandled exception in method `VLEGetAnnotations.getData` brings the program pointer back to method `VLEGetAnnotations.doGet`, and then this method propagates the exception to method `dispatch`. Usually in these situations, the web server sends the stack trace to the client and continues responding to other requests. While the database connection remains open, the next request arrives and opens a second connection by calling method `VLEPostAnnotations.doPost`.

We eliminate the behaviour in the counterexample by changing the `doGet` method of class `VLEGetAnnotations` to the code shown in Figure 10. In the new implementation of method `doGet` we added a try–catch block to catch any exception that may be raised during the execution of method `getData` and to close the connection to the database.

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
                  throws ServletException, IOException {
  createConnection();
  try {
    getData(request, response);
  } catch (Exception ex) { }
  shutdown();
}
```

Fig. 10: The new implementation of method `doGet` of `VLEGetAnnotations` class

We use ProMoVer to re–verify the new code. ProMoVer detects the small change and therefore only re-extracts the specification of method `doGet` of class `VLEGetAnnotations`. It uses this new specification to construct a new PDS and to perform the global model checking. The statistics of the verification are shown in the second row of Table 6. Again, the result of verification is "NO". The provided counterexample is analogous to the previous one; this time, however, the first connection is obtained in method `VLEPostAnnotations.doPost` and the second one in `VLEPostFlag.doPost`. This shows that the same problem of unhandled runtime exceptions exists in the `VLEPostAnnotations.doPost` method, too. In fact, we realized that the same problem exists in some other classes as well. After changing all of them, we use ProMoVer to verify the program, and finally the verification result is "YES".

Code Evolution Scenario. As mentioned above, the Sail–Web project is an ongoing project divided into two main packages. One of the packages includes more features and utilities that probably will be imported into the smaller package in the future. We act proactively and import the missing features and utilities, which we have specified before, into the limited package and form the so called extended package. As mentioned above, this new part consists of 12 servlet classes (called additional classes), which are implemented by 66 public methods with 4806 lines of Java code.

We now use ProMoVer to verify the code of the extended package. The proof storage and reuse mechanism detects the new code, and for each method of the additional classes, checks that its implementation matches the corresponding specification. As shown in the third row of Table 6, the verification takes 32 seconds. This time is spent for local model checking only. The result shows that all new methods respect their specifications. Therefore the verification result is "YES".

7 Conclusion

This paper describes ProMoVer, a tool that supports automatic procedure–modular verification of control flow safety properties of sequences of method invocations. It essentially implements a particular verification scenario for the

CVPP tool set that supports compositional verification of programs with procedures [13]. ProMoVer takes as input a Java program annotated with temporal correctness assertions. The assertions can be written in different specification formalisms. Currently LTL and safety automata are supported.

Modularity is understood here as the relativization of global program correctness properties on the correctness of its components. This is seen as the key to program verification in the presence of static and/or dynamic variability due to code evolution, code customization for many users such as in software product lines (as illustrated in [26]), or as yet unknown or unavailable code such as mobile code. We illustrate two important points: (*i*) temporal safety properties provide a meaningful abstraction for individual methods; and (*ii*) procedure–modular verification of temporal safety properties can be performed automatically. Different specification formalisms can be used to specify those temporal safety properties. Moreover, ProMoVer implements a mechanism for proof storage and reuse, so that only relevant parts have to be reverified after a system change. This makes the verification method advocated by ProMoVer suitable to be used in a context where systems evolve frequently, as is the case e.g., for mobile code. The modularity of the verification allows an independent evolution of the implementations of the individual methods, only requiring the re–establishment of their local correctness.

We believe that writing properties at the procedure–level is intuitive for a programmer. Still, to decrease the effort of annotating programs, we provide support for specification extraction in the case of post–hoc specification of already implemented methods, an inlining–based private method abstraction that requires only public methods to be specified, and a library of standard global safety properties.

Experiments with realistic Java Card and web–based applications show that useful safety properties of such programs can be conveniently expressed in a light–weight notation and verified automatically with ProMoVer. Moreover, proof storage and reuse provide appropriate support for the modular nature of the verification work: local changes in the specification or code require only local re–verification, with significant reduction in verification time.

The addition of safety automata as a specification formalisms has proven to be convenient, and moreover, it also results in more efficient maximal flow graph construction. Still, some issues remain to be resolved in order to increase the utility of ProMoVer. In the future, we plan to also experiment with other temporal logics and notations, or to use patterns to abbreviate common specification idioms. The tool set will be extended with further translations into the underlying uniform logic, which is currently the safety fragment of the modal μ-calculus.

Many important safety properties require program *data* to be taken into account. As a first step towards handling data, work has begun on extending our verification framework and tool set to Boolean programs. We are also currently investigating how to generalize our method for the program model of Rot *et al.* that models object references in the presence of unbounded object creation [24].

Acknowledgments We are indebted to Wojciech Mostowski, Erik Poll and Roberto Guanciale for their help in finding suitable case studies, to Afshin Amighi and Pedro de Carvalho Gomes for helping with the implementation of CVPP and ProMoVer, and to Stefan Schwoon for adapting the input language of Moped to our needs.

References

1. R. Alur, M. Arenas, P. Barcelo, K. Etessami, N. Immerman, and L. Libkin. First-order and temporal logics for nested words. In *Logic in Computer Science (LICS '07)*, pages 151–160, Washington, DC, USA, 2007. IEEE Computer Society.
2. R. Alur and S. Chaudhuri. Temporal reasoning for procedural programs. In *Verification, Model Checking, and Abstract Interpretation (VMCAI '10)*, volume 5944 of *LNCS*, pages 45–60. Springer, 2010.
3. R. Alur, K. Etessami, and P. Madhusudan. A temporal logic for nested calls and returns. In *Tools and Algorithms for the Analysis and Construction of Software (TACAS '04)*, volume 2998 of *LNCS*, pages 467–481. Springer, 2004.
4. A. Amighi, P. de Carvalho Gomes, D. Gurov, and M. Huisman. Sound control-flow graph extraction for java programs with exceptions. In *Software Engineering and Formal Methods (SEFM '12)*, volume 7504 of *LNCS*, pages 33–47, 2012. QC 20121213.
5. O. Burkart, D. Caucal, F. Moller, and B. Steffen. Verification on infinite structures. In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebra*, pages 545–623. North Holland, 2000.
6. R. Cleaveland, J. Parrow, and B. Steffen. A semantics based verification tool for finite state systems. In *International Symposium on Protocol Specification, Testing and Verification*, pages 287–302. North-Holland Publishing Co., 1990.
7. M. Dam. CTL* and ECTL* as fragments of the modal μ-calculus. In *Colloquium on Trees in Algebra and Programming, (CAAP '92)*, volume 581 of *LNCS*, pages 145–164. Springer, 1992.
8. M. Das, S. Lerner, and M. Seigle. ESP: Path–sensitive program verification in polynomial time. In *Programming Language Design and Implementation (PLDI '02)*, pages 57–68. ACM, 2002.
9. Doclet overview. `http://java.sun.com/j2se/1.3/docs/tooldocs/javadoc/overview.html`.
10. N. Gawell. Automatic verification of applet interaction properties. Master's thesis, KTH Royal Institute of Technology, Stockholm, Sweden, 2009. Ref.: TRITA-CSC-E 2009:128.
11. M. Goldman and S. Katz. MAVEN: Modular aspect verification. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '07)*, volume 4424 of *LNCS*, pages 308–322. Springer, 2007.
12. D. Gurov and M. Huisman. Reducing behavioural to structural properties of programs with procedures. In *Verification, Model Checking, and Abstract Interpretation (VMCAI '09)*, volume 5403 of *LNCS*, pages 136–150. Springer, 2009.
13. D. Gurov, M. Huisman, and C. Sprenger. Compositional verification of sequential programs with procedures. *Information and Computation*, 206(7):840–868, 2008.
14. E. Hubbers and E. Poll. Transactions and non-atomic API methods in Java Card: specification ambiguity and strange implementation behaviours. Technical Report NIII-R0438, Radboud University Nijmegen, 2004.

15. L. Hubert, N. Barré, F. Besson, D. Demange, T. Jensen, V. Monfort, D. Pichardie, and T. Turpin. Sawja: Static Analysis Workshop for Java. In *Formal Verification of Object–Oriented Software (FoVeOOS '10)*, volume 6528 of *LNCS*. Springer, 2010.
16. M. Huisman, I. Aktug, and D. Gurov. Program models for compositional verification. In *International Conference on Formal Engineering Methods (ICFEM '08)*, volume 5256 of *LNCS*, pages 147–166. Springer, 2008.
17. M. Huisman and D. Gurov. CVPP: A tool set for compositonal verification of control-flow safety properties. In *Formal Verification of Object–Oriented Software (FoVeOOS '10)*, volume 6528 of *LNCS*, pages 107–121. Springer, 2010.
18. S. Kiefer, S. Schwoon, and D. Suwimonteerabuth. Moped - a model-checker for pushdown systems. http://www.informatik.uni-stuttgart.de/fmi/szs/tools/moped/.
19. D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
20. K. Larsen. Modal specifications. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 232–246. Springer, 1989.
21. G. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, and P. Chalin. *JML Reference Manual*, Feb. 2007. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`.
22. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer, 2002.
23. A. Pnueli. The temporal logic of programs. In *IEEE Symposium on Foundations of Computer Science (FOCS '77)*, pages 46–57. IEEE Computer Society, 1977.
24. J. Rot, F. de Boer, and M. Bonsangue. A pushdown system representation for unbounded object creation. In *Informal pre-proceedings of Formal Verification of Object–Oriented Software (FoVeOOS '10)*, 2010.
25. Sail-web application, 2012. `https://code.google.com/p/sail-web/`.
26. I. Schaefer, D. Gurov, and S. Soleimanifard. Compositional algorithmic verification of software product lines. In *Formal Methods for Components and Objects (FMCO '10)*, volume 6957 of *LNCS*, pages 184–203. Springer, 2011.
27. F. B. Schneider. Enforceable security policies. *ACM Trans. Infinite Systems Security*, 3(1):30–50, 2000.
28. S. Soleimanifard, D. Gurov, and M. Huisman. ProMoVer web interface. `http://www.csc.kth.se/~siavashs/ProMoVer`.
29. S. Soleimanifard, D. Gurov, and M. Huisman. ProMoVer: Modular verification of temporal safety properties. In G. Barthe, A. Pardo, and G. Schneider, editors, *Software Engineering and Formal Methods (SEFM '11)*, volume 7041 of *LNCS*, pages 366–381. Springer, 2011.
30. C. Stirling. *Modal and Temporal Logics of Processes*. Springer, 2001.