ProMoVer: Modular Verification of Temporal Safety Properties[⋆]

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

[1] 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, and is based here 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. 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 smart card domain.

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 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.

[⋆] 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.

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.*, [18]. 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 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.

To support our approach, we have developed a fully automated verification tool, ProMoVer, which can be tried via a web–based interface [20]. 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 [13], to perform the individual local and global correctness checks. 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 applied automatically. 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. 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 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. The ProMoVer web interface allows the user to verify such properties, for which a ready–made formalization is 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 is imposed by the maximal model construction that is the core of our modular verification technique (see [9] for a proof of soundness and completeness for this program model). Still, many useful properties can be expressed at this level of abstraction. These include platform–specific security properties as discussed above, and application–specific 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 generalisation 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.

Control flow safety properties can be expressed in various formalisms, *e.g.*, automata–based or process–algebraic notations, as well as in temporal logics such as LTL [22] and the safety fragment of the modal μ-calculus [15]. Internally, CVPP uses the latter, but ProMoVer allows the user to write the specifications in LTL, which is usually considered more intuitive. It is future work to extend ProMoVer also with other notations, in particular graphical ones.

ProMoVer currently handles procedure–modular verification of control–flow properties for sequential programs. The restriction to modularity at procedure level is 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 modelling multi–threaded programs has been developed earlier (see [12]), 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 one, known theoretical bottlenecks are the maximal model construction and model checking of global properties (both are 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. As to the second limitation, to reduce the effort needed to write specifications, ProMoVer provides a library of common platform-specific global properties, and can extract specifications from a given implementation, as explained above.

The work in this paper is closely related to the development of CVPP [13]. 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 proof reuse, and specification extraction, a collection of ready–formalised properties, and translates between the different intermediate formats and formalisms. Preliminary results on an earlier version of ProMoVer were reported at a workshop [21]. The present paper extends and completes this work. In particular, we have added several facilities to improve the usability of the tool, in the form of automated support for proof reuse, specification extraction, and private method abstraction (see Section 4). Furthermore, we have adapted and extended significantly the experimental evaluation of the tool (see Section 5).

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 [5]. 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 [7]. 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 recapitulates the verification framework, describing the underlying program model and logic, and the compositional verification method based on constructing maximal models. Then, Section 4 describes the ProMoVer tool, while Section 5 describes three small but 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.* [17]). ProMoVer is procedure–modular in the sense that correctness of the global program property is relativized on the local properties of the individual methods. Thus, the overall verification task divides into two independent subtasks:

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

Notice that the second subtask only relies on the local properties 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 property. If the second subtask fails, ProMoVer provides a counter example in the form of a program behaviour that violates the respective property.

```
// @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_ltl_prop: G (X (!odd || !entry) && (even -> X G odd)) */
   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, and every method is annotated with a local property and an interface specifying the required methods. As mentioned above, the interfaces can be extracted from the method implementations. The local method specifications also can be extracted by ProMoVer, see Section 4.

Here we give an intuitive description of the properties specified in the example; a formal definition of the temporal logic LTL is given below in Definition 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 symmetric.

As explained above, the annotated program is correct if (i) methods `even` and `odd` meet their respective local properties, and (ii) the composition of local properties 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 properties. The local properties do not have to be reverified. The verification of the global property fails. As a counter example, ProMoVer returns the following program execution that is allowed by the local properties, but violates the global one:

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

adapted for user understandability by replacing program points with the names of the methods they belong to (*cf.* Definition 3).

3 Framework for Modular Specification and Verification

Next, we briefly present the formal framework underlying the ProMoVer tool that supports this style of procedure–modular verification. It is heavily based on our earlier work on compositional verification [9,8].

3.1 Program Model and Logic

First, we formally define the program model and property specification logic.

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.*

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 2 (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 $\lambda_m : V_m \rightarrow \mathcal{P}(A_m)$ 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 points. 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 3.2).

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.

Flow graph *behavior* is also defined as an instance of an initialized model, induced through the flow graph structure. We use transition label τ for internal transfer of control, m_1 call m_2 for the invocation of method m_2 by method m_1 when method m_2 is provided by the program and m_1 call! m_2 when method m_2 is external, and m_2 ret m_1 respectively m_2 ret? m_1 for the corresponding return from the call.

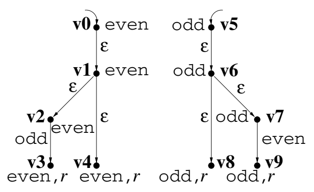

Fig. 2. Flow graph of EvenOdd

Definition 3 (Behavior). *Let $\mathcal{G} = (\mathcal{M}, E) : (I^+, I^-)$ be a flow graph such that $\mathcal{M} = (V, L, \to, A, \lambda)$. The behaviour 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 $S_b = (V \cup I^-) \times V^*$, i.e., states are pairs of control points v or required method names m, and stacks σ, $L_b = \{m_1\ k\ m_2 \mid k \in \{\mathsf{call}, \mathsf{ret}\}, m_1, m_2 \in I^+\} \cup \{m_1\ \mathsf{call!}\ m_2 \mid m_1 \in I^+, m_2 \in I^-\} \cup \{m_2\ \mathsf{ret?}\ m_1 \mid m_1 \in I^+, m_2 \in I^-\} \cup \{\tau\}$, $A_b = A$, $\lambda_b((v, \sigma)) = \lambda(v)$ and $\lambda_b((m, \sigma)) = m$, and $\to_b \subseteq S_b \times L_b \times S_b$ is defined by the following rules:*

$$
\begin{array}{rll}
\text{[transfer]} & (v,\sigma)\xrightarrow{\tau}(v',\sigma) & \text{if } m \in I^+,\ v\xrightarrow{\varepsilon}_m v',\ v \models \neg r \\
\text{[call]} & (v_1,\sigma)\xrightarrow{m_1\ \mathsf{call}\ m_2}(v_2, v_1' \cdot \sigma) & \text{if } m_1, m_2 \in I^+, v_1 \xrightarrow{m_2}_{m_1} v_1', v_1 \models \neg r, \\
 & & \quad v_2 \models m_2, v_2 \in E \\
\text{[ret]} & (v_2, v_1 \cdot \sigma)\xrightarrow{m_2\ \mathsf{ret}\ m_1}(v_1,\sigma) & \text{if } m_1, m_2 \in I^+, v_2 \models m_2 \wedge r, v_1 \models m_1 \\
\text{[call!]} & (v_1,\sigma)\xrightarrow{m_1\ \mathsf{call!}\ m_2}(m_2, v_1' \cdot \sigma) & \text{if } m_1 \in I^+, m_2 \in I^-, v_1 \xrightarrow{m_2}_{m_1} v_1', v_1 \models \neg r \\
\text{[ret?]} & (m_2, v_1 \cdot \sigma)\xrightarrow{m_2\ \mathsf{ret?}\ m_1}(v_1,\sigma) & \text{if } m_1 \in I^+, m_2 \in I^-, v_1 \models m_1
\end{array}
$$

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

Notice that return transitions always hand back control to the caller of the method. Calls to external methods are modeled with intermediate state, from which only an immediate return is possible. In this way possible callbacks from external methods are not captured in the behaviour. This simplification is justified, since we abstract away from data in the model and the behaviour is thus context–free, but has to be kept in mind when writing specifications; in particular one cannot specify that callbacks are not allowed.

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

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

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 behaviour is the following run:

$$(v_0,\varepsilon)\xrightarrow{\tau}(v_1,\varepsilon)\xrightarrow{\tau}(v_2,\varepsilon)\xrightarrow{\text{even call! odd}}(\texttt{odd},v_3)\xrightarrow{\text{odd ret? even}}(v_3,\varepsilon)$$

Pushdown systems. (PDS) are an alternative way to express flow graph behaviour. We exploit this by using PDS model checking, concretely the tool MOPED [14], for verifying program behaviour against temporal formulas.

As mentioned above, safety properties can be expressed in many different formalisms. In this paper, we use *safety LTL* which consists of the safety–fragment of *Linear Temporal Logic* (LTL), using the weak until–operator. Internally, however, the whole machinery is based on the safety fragment of the modal μ-calculus. Safety LTL is somewhat less expressive than the latter and can be uniformly encoded in it. This translation is implemented as part of PROMOVER. In our LTL formulas, we use an additional atomic proposition `entry` that holds for entry nodes. It is removed by the translation into the modal μ-calculus.

Definition 4 (Safety LTL). *Let* $p \in A_b \cup \{\texttt{entry}\}$ *and* $m \in M$*. The formulae of* Safety LTL *are inductively defined by:*

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

Satisfaction on states $(\mathcal{M}_b, s) \models \phi$ for LTL is defined in the standard fashion [22]: 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 `&&` is ASCII notation for $\wedge$) and its intuitive meaning in Example 1. Flow graph extraction and construction ensures 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.

3.2 Compositional Verification

Our method for *algorithmic compositional verification* 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 [9].

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, that 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.

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

As mentioned above, 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 flow 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 $\biguplus_{i=1,\ldots,k} \mathcal{M}ax(\psi_i, I_i) \models \phi$:** (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 behaviour of $\mathcal{G}_{\mathcal{M}ax}$ as a PDS and use a standard PDS model checker.

Example 6. Consider again the annotated Java program from Example 1. ProMoVer first extracts the method flow 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 $\mathcal{G}_{odd} \models \psi_{odd}$ by standard finite state model checking. Independently, it constructs the maximal flow graphs of methods `even` and `odd`, denoted $\mathcal{M}ax(\psi_{even}, I_{even})$ and $\mathcal{M}ax(\psi_{odd}, I_{odd})$, respectively, and composes the graphs to obtain $\mathcal{G}_{\mathcal{M}ax} = \mathcal{M}ax(\psi_{even}, I_{even}) \uplus \mathcal{M}ax(\psi_{odd}, I_{odd})$. Finally, ProMoVer translates $\mathcal{G}_{\mathcal{M}ax}$ to a PDS and model checks the latter against the global property.

4 The ProMoVer Tool

Next we describe the internals of ProMoVer. As mentioned above, ProMoVer essentially is a wrapper for CVPP [13], 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 [20].

CVPP Wrapper. Figure 3 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 [6], and then passes properties and interfaces on to the different CVPP tools.

Task (i) first invokes the Analyzer to extract the method graphs of the program. This builds on Sawja [11] to extract flow graphs from Java bytecode. Then

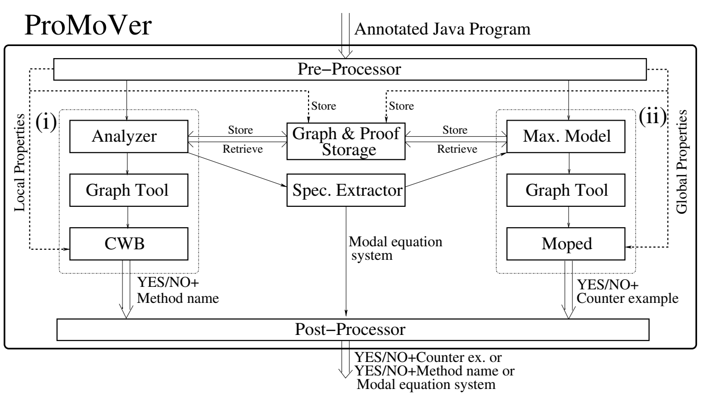

Fig. 3. Overview of ProMoVer and its underlying tool set

the Graph Tool is used. This implements several algorithms on flow graphs, including flow graph composition $\uplus$ and translations of flow graphs into different formats. Here the Graph Tool is used to translate the flow graph of each method into a CCS model. These are then model checked against the respective local method specifications using the *Concurrency Workbench* (cwb) [4].

Task (*ii*) first constructs a maximal flow graph for every method using the Maximal Model Tool. Then the Graph Tool composes the generated flow graphs and converts the result into a PDS. Finally Moped [14] is used to model check the PDS against the global property.

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, the counter example provided by Moped, transformed into a program execution, 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 the form of modal equation systems (as defined by Larsen [16]). These are equivalent to formulae in modal μ-calculus with boxes and greatest fixed points only, and have the advantage that in cvpp they can serve directly as input for the construction of maximal flow graphs. It is future work to also extract to other specification languages, such as LTL.

Consider again Figure 1. Specification extraction for method `even` results in (where `eps` is ASCII notation for ε, and `ff` denotes *false*):

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

This specifies that method `odd` may be called at most once: initially `X0` holds, and method `odd` may be called or an internal step (labelled `eps`) may be made. After calling `odd`, `X1` should hold and only internal steps are allowed.

As a more involved example, consider method `m` and its extracted specification:

```
@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. Actually, the order of invoking `m1` and `m2` is immaterial for this program, so 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;
```

Private Method Abstraction. Since private methods are used as 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 behaviour, and users only have to specify the public methods. For details the reader is referred to [9].

Property Specification Library. ProMoVer's web interface provides a collection of pre–formalised 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.

5 Experimental Results with ProMoVer

We use ProMoVer to verify a standard control flow safety property on a number of Java Card applications. Java Card is one of the leading interoperable platforms for smart cards. Many smart card applications are security–critical.

Table 1. Applications details

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

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. Safe use of it demands that certain methods are not called 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.

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 [10] – 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`. Typical Java Card programming standards, such as the Global Platform specification, state that these methods may not be used within a transaction. We use ProMoVer to verify that applications comply with this *Safe 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

Table 2. Verification Results

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

network service. It contains methods to look up and to 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 information 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 Safe 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:

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

Extracting Local Method Specifications. The specification extractor is used to obtain local specifications for every public method. Basically, these describe the order of method invocations. We inspected those for immaterial orderings, and translated the adjusted representations into safety LTL. The intention is that local method specifications capture the allowed sequences of method calls made from within the specified method, but in an abstract way, allowing for possible evolution of the method implementations.

Verification Results. After annotating the applications, they are passed to ProMoVer. The tool extracts the flow graph of the applications, and partitions them into the individual method graphs to verify adherence to the local properties. Further, for each local property a maximal flow graph is constructed, and

Table 3. Proof Reuse Results

	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

their composition is verified *w.r.t.* the global property above. The statistics for these verifications are given summarized in Table 2. The table shows: the time spent by the pre–processor (PPT) and the graph extractor (GE), 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 times are in seconds, and were obtained on a SUN SPARC machine.

We also experimentally evaluated the advantages of exploiting the proof storage and reuse mechanism. After the first verification, when method and maximal flow graphs are stored, for each application, we once changed the source code and once the local specification of a public method, and used ProMoVer to reverify the application. The result of proof reuse are shown in Table 3. The numbers show that proof reuse can reduce significantly the verification time for larger applications.

6 Conclusion

This paper describes ProMoVer, a tool that supports automatic procedure–modular verification of control flow safety properties of sequences of method invocations. ProMoVer takes as input a Java program annotated with temporal correctness assertions. It essentially implements a particular verification scenario for the cvpp tool set that supports compositional verification of programs with procedures [9].

Modularity is understood here as the relativization of global program correctness properties on the correctness of its components, and 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, 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. 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 software product lines or 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 applications show that useful safety properties of such programs can be conveniently expressed in a light–weight notation and verified automatically with ProMoVer.

Still, some issues remain to be resolved in order to increase the utility of ProMoVer. Both for pre– and post–hoc method specification, notations based on automata or process algebra may prove more convenient than LTL, and may also allow more efficient maximal flow graph construction. Ultimately, our goal is that all specifications (local and global) can be written in various temporal logics and notations, or to use patterns to abbreviate common specification idioms. The tool set will provide translations into the underlying uniform logic, which is currently the safety fragment of the modal μ-calculus. However, because of limitations on the currently available PDS model checkers, global properties have at present to be written in LTL.

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 [19].

Finally, to investigate the *scalability* of the approach, we plan to perform a significantly larger case study.

Acknowledgments. We are indebted to Wojciech Mostowski and Erik Poll for their help in finding a suitable case study, 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. Alur, R., Arenas, M., Barcelo, P., Etessami, K., Immerman, N., Libkin, L.: First-order and temporal logics for nested words. In: Logic in Computer Science (LICS 2007), pp. 151–160. IEEE Computer Society, Washington, DC, USA (2007)
2. Alur, R., Chaudhuri, S.: Temporal reasoning for procedural programs. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 45–60. Springer, Heidelberg (2010)
3. Alur, R., Etessami, K., Madhusudan, P.: A temporal logic of nested calls and returns. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 467–481. Springer, Heidelberg (2004)

4. Cleaveland, R., Parrow, J., Steffen, B.: A semantics based verification tool for finite state systems. In: International Symposium on Protocol Specification, Testing and Verification, pp. 287–302. North-Holland Publishing Co., Amsterdam (1990)
5. Das, M., Lerner, S., Seigle, M.: ESP: Path–sensitive program verification in polynomial time. In: Programming Language Design and Implementation (PLDI 2002), pp. 57–68. ACM, New York (2002)
6. Doclet overview, `http://java.sun.com/j2se/1.3/docs/tooldocs/javadoc/overview.html`
7. Goldman, M., Katz, S.: MAVEN: Modular aspect verification. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 308–322. Springer, Heidelberg (2007)
8. Gurov, D., Huisman, M.: Reducing behavioural to structural properties of programs with procedures. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 136–150. Springer, Heidelberg (2009)
9. Gurov, D., Huisman, M., Sprenger, C.: Compositional verification of sequential programs with procedures. Information and Computation 206(7), 840–868 (2008)
10. Hubbers, E., Poll, E.: Transactions and non-atomic API methods in Java Card: specification ambiguity and strange implementation behaviours. Technical Report NIII-R0438, Radboud University Nijmegen (2004)
11. Hubert, L., Barré, N., Besson, F., Demange, D., Jensen, T., Monfort, V., Pichardie, D., Turpin, T.: Sawja: Static Analysis Workshop for Java. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 92–106. Springer, Heidelberg (2011)
12. Huisman, M., Aktug, I., Gurov, D.: Program models for compositional verification. In: Liu, S., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 147–166. Springer, Heidelberg (2008)
13. Huisman, M., Gurov, D.: CVPP: A tool set for compositional verification of control–flow safety properties. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 107–121. Springer, Heidelberg (2011)
14. Kiefer, S., Schwoon, S., Suwimonteerabuth, D.: Moped - a model-checker for push-down systems, `http://www.informatik.uni-stuttgart.de/fmi/szs/tools/moped/`
15. Kozen, D.: Results on the propositional μ-calculus. Theoretical Computer Science 27, 333–354 (1983)
16. Larsen, K.: Modal specifications. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 232–246. Springer, Heidelberg (1990)
17. Leavens, G., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P.: JML Reference Manual, Department of Computer Science, Iowa State University (February 2007), `http://www.jmlspecs.org`
18. Müller, P.: Modular Specification and Verification of Object-Oriented Programs. LNCS, vol. 2262. Springer, Heidelberg (2002)
19. Rot, J., de Boer, F., Bonsangue, M.: A pushdown system representation for unbounded object creation. In: Informal pre-proceedings of Formal Verification of Object–Oriented Software (FoVeOOS 2010) (2010)
20. Soleimanifard, S., Gurov, D., Huisman, M.: ProMoVer web interface, `http://www.csc.kth.se/~siavashs/ProMoVer`
21. Soleimanifard, S., Gurov, D., Huisman, M.: Procedure–modular verification of control flow safety properties. In: Workshop on Formal Techniques for Java Programs, FTfJP 2010 (2010)
22. Stirling, C.: Modal and Temporal Logics of Processes. Springer, Heidelberg (2001)