Procedure-Modular Verification of Control Flow Safety Properties*

Siavash Soleimanifard
Royal Institute of Technology
Stockholm, Sweden
siavashs@csc.kth.se

Dilian Gurov
Royal Institute of Technology
Stockholm, Sweden
dilian@csc.kth.se

Marieke Huisman
University of Twente
Enschede, Netherlands
Marieke.Huisman@ewi.utwente.nl

ABSTRACT

This paper describes a novel technique 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 of verification is achieved by relativizing the correctness of global properties on the local properties rather than on the implementations of methods, and is based on the construction of maximal models. Tool support is provided by means of ProMoVer, a tool that is essentially a wrapper around a previously developed tool set for compositional verification of control flow safety properties, where program data is abstracted away completely. We evaluate the technique on a small but realistic case study.

Keywords

Modular Verification, Temporal Logic, Maximal Models.

1. INTRODUCTION

Modularity is a design paradigm that aims at controlling the complexity of developing large software and facilitates the reuse of components. When applied to *verification*, i.e., to establish the formal correctness of a software product, modularity requires that correctness of the software modules (components) is specified and verified independently (*locally*) for each module, while the 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. Such an approach allows an independent evolution of the implementations of individual modules, only requiring the re–establishment of their local correctness (provided the local specifications have not changed).

*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.

FTfJP '10, June 22, 2010, Maribor, Slovenia

Hoare logic provides one popular framework for modular specification and verification of software. For procedural programming languages it is natural to take the individual procedures as modules, in order to achieve scalability. In static checkers such as ESC/Java [6], one provides as input a program annotated with specification assertions - as comments ignored by the compiler - and the tool then automatically checks the code against the specification, assuming that called methods respect their specification.

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 appropriate also for this second class of correctness properties.

We have developed an automated verification tool, ProMoVer, that takes as input a Java program annotated with global and method–local correctness assertions written in temporal logic, and automatically invokes a number of tools from a previously developed tool set for compositional verification [9] to perform the individual local and global correctness checks[1]. Essentially, ProMoVer is a wrapper that automatically performs a standard verification scenario in the general tool set. Validity of the approach is shown on an electronic purse application that is used for securely transferring money. Such types of *security relevant* applications are an important target for formal verification techniques.

To allow efficient algorithmic modular verification, the tool set currently abstracts away from all data. In particular, method calls in Java programs are approximated by a non–deterministic choice on possible method implementations that the virtual call resolution might resolve to. This may seem like a severe restriction, but still many useful properties can be expressed. Section 5 shows how the tool set is used to show the absence of non–atomic methods within a Java Card transaction (i.e., a mechanism to guarantee atomic updates). Other useful properties that can be expressed and verified in our framework are for example:

- a given method that changes certain sensitive data is only called from within another dedicated authentication method, i.e., unauthorized access is not possible;
- before program state is being dumped into memory, a serialization method is called to arrange the state;
- in a voting system, candidate selection has to be finished, before the vote can be confirmed;

[1] ProMoVer is available via web–based interface [15]

- in a door access control system, the password has to be checked before the door is unlocked, and the password can only be changed if the door is unlocked.

Extending the technique with data over finite domains will allow for a wider range of properties and possible applications, but needs to be combined with abstraction techniques to control the complexity of verification. Such an extension will be investigated in future work.

Overview.

Section 2 presents the tool from a user's point–of–view. The next section summarizes our 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 a small but realistic case study using the tool. Finally, the last section draws conclusions and suggests directions for future research.

Related Work.

As already pointed out, the present work is based on a previously developed method and tool set for compositional verification of control flow safety properties [9, 8]. Essentially, we provide a wrapper around the tool set to support the fully automated procedure–modular verification of programs annotated with temporal correctness assertions.

A non–compositional verification method based on a program model closely related to ours is presented by Alur and others [4]. It proposes a temporal logic CaRet for nested calls and returns (generalized to a logic for nested words in [2]) that can be used to specify regular properties of local paths within a procedure that skips over calls to other procedures.

Maven is a modular verification tool addressing temporal properties of procedural languages, but in the context of aspects [7].

Müller was the first to propose a sound modular Hoare–style verification technique for object–oriented languages [14]. 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 [3].

2. THE TOOL: A USER'S VIEW

The goal of the present work is the development of a fully automated tool for procedure–modular verification of control flow safety properties of Java programs. These properties are provided to the tool as assertions in the form of program annotations: programs are annotated with global program properties and methods are annotated with local method properties. We use a JML–like syntax for annotations (*cf.* [13]), and similarly present them as special comments. The tool checks (*i*) whether the method implementations respect the local properties, and (*ii*) whether the local properties are sufficient to ensure the global property. If this is not the case, a counter example is provided in the form of a program behavior that violates the respective property.

Our approach 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 naturally divides into two independent subtasks:

(*i*) check that each method implementation satisfies its local property, and

(*ii*) 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.

Control flow safety properties can be expressed in automata–based or process–algebraic notations, as well as in temporal logics such as LTL [16] and the safety fragment of the modal μ-calculus [12]. Our tool currently supports the latter two notations, as illustrated by the example below.

In addition to the properties, the tool also requires global and local *interfaces*. A global interface consists of a list of all 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). The user only has to specify the local required methods, the global interface can be deduced from the local method information.

```
/** @global_LTL_prop:
 *    even -> X ((even && !entry) W odd)
 */
public class EvenOdd {
   /** @local_interface: requires {odd}
    *
    *  @local_prop:
    *   nu X1. (([even call even]ff) /\ ([tau]X1) /\
    *     [even caret odd] nu X2.
    *       (([even call even]ff) /\
    *        ([even caret odd]ff) /\ ([tau]X2))
    */
   public boolean even(int n) {
      if (n == 0) return true;
      else return odd(n-1);
   }

   /** @local_interface: requires {even}
    *
    *  @local_prop:
    *    nu X1. (([odd call odd]ff) /\ ([tau]X1) /\
    *       [odd caret even] nu X2.
    *         (([odd call odd]ff) /\
    *          ([odd caret even]ff) /\ ([tau]X2))
    */
   public boolean odd(int n) {
      if (n == 0) return false;
      else return even(n-1);
   }
}
```

Figure 1: A simple annotated Java program

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. As mentioned above, the global interface is extracted from the local interfaces. In addition, every method is annotated with a local property and an interface specifying the required methods.*

Definitions 4 and 5 below formally define the logics used to write the specifications; this example just gives the intuitive idea. The global specification is written in LTL and 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` *is written in the safety fragment of the modal μ-calculus and expresses that "method* `even` *can only call method* `odd`*, and after returning from the call, no other method can be called" (where* `ff` *denotes* false*). The local property of method* `odd` *has a similar meaning.*

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 in the previous example we change the global property to "in every program execution starting in method* `even`*, the first call is to method* `even` *itself", the annotated program becomes incorrect, because subtask (ii) fails, as demonstrated by the following program execution:*

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

This execution is allowed by the local properties, but violates the global one. Our tool therefore returns this as a counter example.

Next, we present the formal framework and the PROMOVER tool that support this style of procedure–modular verification.

3. FRAMEWORK

This section outlines the theoretical framework upon which our verification method rests. It is heavily based on our earlier work on compositional verification [9, 8].

3.1 Program Model and Logic

We begin by formally defining the program model and logics for specifying properties.

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 points *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 can have multiple entry points.

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. Interfaces are needed when constructing 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.

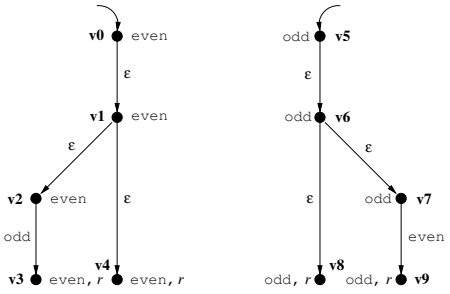

Figure 2: Flow graph of EvenOdd

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.*

We define the *behavior* of a flow graph as a labeled transition system (LTS). 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, m_2 *ret* m_1 for the corresponding return from the call, and label m_1 *caret* m_2 for the (atomic) invocation of and return from an external method m_2 by method m_1.

DEFINITION 3 (**Behavior**). *Let $\mathcal{G} = (\mathcal{M}, E) : (I^+, I^-)$ be a flow graph such that $\mathcal{M} = (V, L, \rightarrow, A, \lambda)$. The behavior of $\mathcal{G}$ is defined as initialized model $b(\mathcal{G}) = (\mathcal{M}_b, E_b)$, where $\mathcal{M}_b = (S_b, L_b, \rightarrow_b, A_b, \lambda_b)$, such that $S_b = V \times V^*$, i.e., states (or configurations) are pairs of control points v and stacks σ, $L_b = \{m_1\ k\ m_2 \mid k \in \{call, ret\},\ m_1, m_2 \in I^+\} \cup \{m_1\ caret\ m_2 \mid m_1 \in I^+ \wedge m_2 \in I^-\} \cup \{\tau\}$, $A_b = A$, $\lambda_b((v, \sigma)) = \lambda(v)$, and $\rightarrow_b \subseteq S_b \times L_b \times S_b$ is defined by the rules:*

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

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

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

[caret] $(v_1, \sigma) \xrightarrow{m_1\ \texttt{caret}\ m_2} (v_1', \sigma)$ 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 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. One example run through its (branching, infinite–state) behavior, from an initial to a final configuration, 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:*

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

An alternative way to express flow graph behavior is to use *pushdown systems* (PDS). This can be exploited by using pushdown system model checking for verifying behavioral properties. In this work, we use PDS to express behaviors and use the tool Moped to model check PDS against temporal formulas [11].

The specification language for local behavioral properties, around which our compositional verification method is centered, is *simulation logic*, the fragment of the modal μ–calculus [12] with boxes and greatest fixed–points only. This temporal logic is adequate for expressing safety properties.

Definition 4 (**Simulation Logic**). *The formulae of* simulation logic *are inductively defined by:*

$$\phi ::= p \mid \neg p \mid X \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid [a]\phi \mid \nu X.\,\phi$$

where $p \in A_b$, $a \in L_b$ and X ranges over propositional variables.

Satisfaction on states $(\mathcal{M}_b, s) \models \phi$ is defined in the standard fashion [12]. For instance, formula $[a]\phi$ holds of state s in model $\mathcal{M}_b$ if ϕ holds in all states accessible from s via an edge labeled a. An initialized model $(\mathcal{M}_b, E_b)$ satisfies a formula ϕ, denoted $(\mathcal{M}_b, E_b) \models \phi$, if all its initial configurations E_b satisfy ϕ. The constant formulae *true* (denoted `tt`) and *false* (`ff`) are definable. For convenience, we use $p \Rightarrow \phi$ to abbreviate $\neg p \vee \phi$.

Safety properties can also be expressed in other formalisms and logics, such as *Linear Temporal Logic* (LTL). In this paper, we use the safety–fragment of LTL that uses the weak version of until.

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

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

where $p \in A_b$.

Satisfaction on states $(\mathcal{M}_b, s) \models \phi$ is again defined in the standard fashion [16]. For instance, formula $\texttt{X}\,\phi$ holds of state s in model $\mathcal{M}_b$ if ϕ holds in the second state of every run starting in s, while $\phi \,\texttt{W}\, \psi$ 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 such that ϕ holds in all previous states.

Example 5. *Consider the global LTL property of class* `EvenOdd` *in Figure 1 and its intuitive meaning given in Example 1. In the formula,* `&&` *and* `!` *are ASCII formats of* $\wedge$ *and* $\neg$ *respectively, while* `entry` *is an atomic proposition that holds at entry nodes of methods. The formula expresses precisely the property "if program execution starts in method* `even`*, then from the next state on, control stays in non–entry points of* `even` *as long as it does not reach method* `odd`*". Assuming entry points are only reachable via calls, this interpretation coincides with the one given in Example 1.*

This fragment of LTL is somewhat less expressive than simulation logic and can be uniformly encoded in it.

3.2 Compositional Verification

Our method for *compositional verification* is based on the construction of maximal flow graphs from component properties. For a given property ψ and interface I, consider the class of all flow graphs with interface I satisfying ψ. A *maximal flow graph* for ψ and I is a flow graph $\mathcal{M}ax(\psi, I)$ that satisfies exactly those properties that hold for all members of the class. Thus, the maximal flow graph can be used as a representative of the class for the purpose of checking properties.

The main principle of compositional verification based on maximal flow graphs can be presented, for a system with k components, as a proof rule with $k+1$ premises:

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

The principle states that the composition of components $\mathcal{G}_1 : I_1, ..., \mathcal{G}_k : I_k$ satisfies a global property ϕ if for some local properties ψ_i satisfied by the corresponding components $\mathcal{G}_i$, the composition of the maximal flow graphs for ψ_i and I_i satisfies property ϕ. For details the reader is referred to [9].

In *procedure–modular verification*, we consider compositional verification on the method level, i.e., we consider the program methods as components. Let M be the set of program methods, where $k = |M|$, and let ψ_i and $\mathcal{C}_i$ be the specification and the implementation of method m_i, respectively. To instantiate the above compositional verification principle to procedure–modular verification, the following two independent tasks have to be performed:

(i) Checking $\mathcal{C}_i \models \psi_i$ for $i = 1, ..., k$.

For each method $m_i \in M$, (1) extract the method flow graph $\mathcal{G}_i$ from $\mathcal{C}_i$, and (2) 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

$$\biguplus_{i=1,\ldots,k} \mathcal{M}ax(\psi_i, I_i) \models \phi$$

(1) Construct the maximal flow graphs $\mathcal{M}ax(\psi_i, I_i)$ for all method specifications ψ_i and interfaces I_i, then (2) compose the graphs, resulting in flow graph $\mathcal{G}_{\mathcal{M}ax}$, and finally (3) model check $\mathcal{G}_{\mathcal{M}ax}$ against the 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.

EXAMPLE 6. *Consider again the annotated Java program from Example 1. Following procedure–modular verification, we first extract the method flow graphs of methods* `even` *and* `odd`*, denoted* $\mathcal{G}_{even}$ *and* $\mathcal{G}_{odd}$*, respectively. Next, we check* $\mathcal{G}_{even} \models \psi_{even}$ *and* $\mathcal{G}_{odd} \models \psi_{odd}$ *by standard finite state model checking. Independently, we construct 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 compose 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, we translate* $\mathcal{G}_{\mathcal{M}ax}$ *to a PDS and model check the latter against the global property.*

4. THE ProMoVer TOOL

This section presents PROMOVER, a tool for procedure–modular verification of annotated Java programs, built on top of a previously developed tool set for compositional verification described in [9]. PROMOVER is implemented in Python and can be tested online [15].

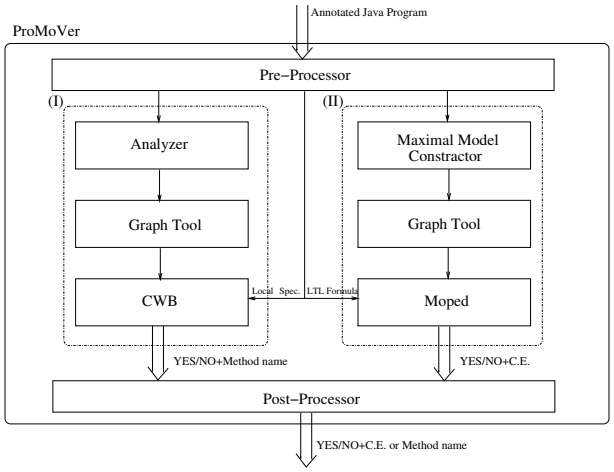

Figure 3: Overview of PROMOVER and its underlying tool set

An overview of the individual tools from the tool set that we have used in this work is shown in Figure 3. In the figure, *Graph Tool* is a collection of algorithms on flow graphs, including flow graph composition $\uplus$ and translations of flow graphs to different formats, used by other parts of the tool set. In particular, we use here the flow graph composer and the translators to CCS terms and PDS.

As input, PROMOVER accepts annotated Java programs as exemplified in Section 2. The *pre–processor* uses the Java Doclet API [1] to parse the annotations of the program and passes the properties and interfaces to the other tools. The two different tasks described in Section 3.2 are performed in different blocks.

Block (I) is related to task (i): We extract the method graphs of the program with the *Analyzer* tool that builds on SOOT [17] to extract flow graphs from Java bytecode. Then, we use the *Graph Tool* to generate a CCS model for every method graph, and we model check these models against the respective method specifications using the *Concurrency Workbench* (CWB) [5].

Block (II) is related to task (ii): We construct a maximal flow graph for every method using the *Maximal Model Constructor*, compose the generated flow graphs and convert the result to a PDS with the *Graph Tool*, and finally use MOPED [11] to model check the PDS against the global LTL property.

The *post–processor* collects all model checking results and converts these into a user–understandable format. It returns a positive result if the result of all collected model checking tasks is positive, and a negative result if at least one of the model checking tasks does not succeed. In the latter case, if one of the model checking tasks from block (I) fails, PROMOVER returns the name of the method that does not satisfy its specification. If it is the model checking task from block (II) that fails, then PROMOVER transforms the counter example provided by MOPED into a program execution and returns this.

5. EXPERIMENTS

This section describes our experience with using PROMOVER to verify control flow safety properties of the JavaPurse application. This is a realistic e-commerce application developed by Sun Microsystems to demonstrate the use of the Java Card environment for developing electronic purse applications[2]. Java Card technology provides a secure environment developed by Sun Microsystems to support applications on smart cards. It is one of the leading interoperable platforms for smart cards.

One of the features of the Java Card environment is the transaction mechanism that ensures that data remains consistent upon a power loss. Safe use of this mechanism demands that certain methods are not called within a transaction. We show how this is expressed and verified for the JavaPurse application.

5.1 The Java Card Transaction Mechanism

Smart cards have two types of writable memory, *persistent memory* (EEPROM or Flash) and *transient memory* (RAM). The Java Card memory model adheres to this. 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 cope with card tears (and related problems, such as resets), Java Card provides a *transaction mechanism*. This 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]. These methods are provided by class `JCSystem` of the Java Card API.

The Java Card API contains some *non-atomic* methods that cannot be used when a transaction is in progress. Notably, the class `javacard.framework.Util`, that provides functionality to store and update byte arrays, contains meth-

[2]The JavaPurse source code can be downloaded from Sun Microsystems, `http://java.sun.com/javacard/`.

ods `arrayCopyNonAtomic` and `arrayFillNonAtomic` that may not be used within a transaction. (For safe array updating within a transaction, the class also provides the atomic method `arrayCopy`.) We show how ProMoVer can be used to verify that applications comply with this policy.

5.2 The JavaPurse Application

JavaPurse is a smart card electronic purse application providing secure money transfers. The application contains a balance record denoting the user's current and maximum credits. It contains methods `processInitializeTransaction` and `processCompleteTransaction` that initialize, perform and complete a secure transaction using the Java Card transaction mechanism. 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. These data updates use the API methods `arrayFillNonAtomic`, `arrayCopyNonAtomic` and `arrayCopy` mentioned above. This functionality of the JavaPurse application is implemented by means of 19 methods with approximately $1K$ lines of code in total.

The JavaPurse application contains 222 method calls, 15 of which are method calls to `arrayCopyNonAtomic` and 6 to `arrayFillNonAtomic`. Transactions are used in two places. One of the transactions contains 2 API method invocations, the other contains 3 API method invocations. Method `abortTransaction` is not used in JavaPurse, and is not considered here.

5.3 Specification of JavaPurse

As mentioned above, we want to ensure formally that the non-atomic methods `arrayCopyNonAtomic` and `arrayFillNonAtomic` are not invoked within a transaction. Hence, we state the following global control flow safety property for JavaPurse:

> 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 with the following LTL formula, with which we annotated the program:

$$\mathtt{G}\,(\mathtt{beginTransaction} \rightarrow \neg\mathtt{NonAtomicOp}\;\mathtt{W}\;\mathtt{commitTransaction})$$

where `NonAtomicOp` can be either `arrayCopyNonAtomic` or `arrayFillNonAtomic`.

Next, we annotated every method of JavaPurse with a local specification consisting of an interface and a formula in simulation logic. The specifications were obtained in a post–hoc manner, after inspecting the code. The intention is that local method specifications capture the allowed sequences of method calls made from within the specified method, but in an abstract away, allowing for possible evolution of the method implementations. The constructed formulae are comparatively long, but follow the same simple pattern as for methods `even` and `odd` in Figure 1. To simplify the specification task, we plan in future work to provide support for appropriate specification patterns.

As discussed below, the global property is then proven on the basis of these local method specifications only, i.e., without looking at the method implementations.

5.4 Verification Results

After annotating the JavaPurse program, it is passed to ProMoVer. The flow graph extracted from the program code as part of verification subtask (i) consists of 1018 nodes and 1128 edges. ProMoVer partitioned this graph to obtain the individual method graphs, and checked each of these against its local specification.

Subtask (ii) ensures that the local method specifications guarantee the global property. First, for each method a maximal flow graph is generated from its local specification. The maximal flow graphs are then composed, and the resulting flow graph, consisting of 200 nodes and 1464 edges, is translated by ProMoVer to a PDS and model checked against the global property.

ProMoVer returns a positive result, meaning that all method implementations in JavaPurse meet their local specifications, and that the method specifications indeed entail the global safety property.

The whole verification is performed by ProMoVer in 150 seconds, running on a SUN SPARC machine. Subtask (i) is performed in about 142 seconds, subtask (ii) in about 4 seconds, and the remaining 4 seconds are spent on pre– and post–processing. This relatively slow performance is largely due to the external tool Soot, part of our *Analyzer*, which needed 141 seconds to extract the control flow graph.

6. CONCLUSION

This paper describes the ProMoVer tool that allows to automatically verify control flow safety properties of sequences of method invocations in a procedure–modular fashion. ProMoVer takes as input a Java program annotated with temporal correctness assertions. The technique builds on a previously developed general method for the compositional verification of programs with procedures [9].

Because of the focus on modularity at the procedure level, we can take advantage of the fact that local properties are relatively small and simple to write. Moreover, this focus also makes checking whether the local properties are sufficient to guarantee the global property efficient. We believe that writing properties at procedure–level is intuitive for a programmer, because this is the level of abstraction at which he thinks about the program. However, we plan to perform a comparison with other, less fine–grained notions of software modules.

The experiment with JavaPurse shows that safety properties of realistic Java programs can be conveniently expressed in a light–weight notation and verified automatically with ProMoVer. Modularity of the approach allows an independent evolution of the implementations of the individual methods, only requiring the re–establishment of their local correctness.

Still, some issues remain to be resolved in order to increase the utility of the tool. If method specifications are to be produced from an implementation, i.e., post–hoc, as in the above exercise, automated support has to be provided to reduce the manual task and the risk for specification errors. For pre–hoc specification, notations based on automata or process algebra may prove more convenient than simulation logic, and may also allow more efficient maximal flow graph construction.

As explained, currently local method specifications are

written in simulation logic, and global properties in weak LTL. Ultimately, our goal is that all specifications (local and behavioral) can be written in various temporal logics and different notations, e.g., based on automata, or using patterns to abbreviate common specification idioms. The tool set should then provide translations into a common logic where necessary. So far, for the development of the tool set our focus has been on simulation logic, because of its expressiveness. However, because of limitations on the current PDS model checkers, global properties have to be written in weak LTL.

Many interesting 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.

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

Acknowledgment.

We are indebted to Wojciech Mostowski and Erik Poll for their help in finding a suitable case study, and to Stefan Schwoon for adapting the input language of the PDS model checker MOPED to our needs.

7. REFERENCES

[1] Doclet overview. `http://java.sun.com/j2se/1.3/docs/tooldocs/javadoc/overview.html`.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '02)*, pages 234–245, 2002.

[7] 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.

[8] 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.

[9] D. Gurov, M. Huisman, and C. Sprenger. Compositional verification of sequential programs with procedures. *Information and Computation*, 206(7):840–868, 2008.

[10] E. Hubbers and E. Poll. Transactions and non-atomic api methods in java card: specification ambiguity and strange implementation behaviours.

[11] S. Kiefer, S. Schwoon, and D. Suwimonteerabuth. Moped - a model-checker for pushdown systems. `http://www.informatik.uni-stuttgart.de/fmi/szs/tools/moped/`.

[12] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[13] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. 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`.

[14] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer, 2002.

[15] S. Soleimanifard, D. Gurov, and M. Huisman. PROMOVER, a tool for procedure–modular verification: A tool web interface. `http://www.csc.kth.se/~siavashs/ProMoVer/promover.php`.

[16] C. Stirling. *Modal and Temporal Logics of Processes*. Springer, 2001.

[17] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java Optimization Framework. In *CASCON '99*, pages 125–135, 1999.