Program Models for Compositional Verification[⋆]

Marieke Huisman[1], Irem Aktug[2], and Dilian Gurov[2]

[1] INRIA Sophia Antipolis, France
[2] Royal Institute of Technology, Stockholm, Sweden

Abstract. Compositional verification is crucial for guaranteeing the security of systems where new components can be loaded dynamically. In earlier work, we developed a compositional verification principle for control-flow properties of sequential control flow graphs with procedures. This paper discusses how the principle can be generalised to richer program models. We first present a generic program model, of which the original program model is an instantiation, and explicate under what conditions the compositional verification principle applies. We then present two other example instantiations of the generic model: with exceptional and with multi-threaded control flow, and show that for these particular instantiations the conditions hold. The program models we present are specifically tailored to our compositional verification principle; however, they are sufficiently intuitive and standard to be useful on their own. Tool support and practical application of the method are discussed.

1 Introduction

Compositional verification addresses the problem of proving the correctness of a compound system based on properties of its components. Compositional verification techniques allow one to guarantee that if the new applications satisfy certain local requirements, the global security (policy) of the system is not violated. Such techniques are crucial to ensure the security of any platform, where new applications can be installed dynamically. Typical application areas are *e.g.*, mobile computing, and dynamically reconfiguring distributed systems.

We are interested in both structural and behavioural control flow properties of programs. A structural property is a property of the (finite) flow graph itself, such as "every path from the entry of method m_1 to a call instruction to method m_2 passes a call instruction to method m_3". A behavioural property is a property of the (infinite state) behaviour induced by the flow graph, such as "in any execution of the program, method m_1 calls method m_2 at most once".

In earlier work, we developed a compositional verification method for programs with procedures. Our method supports the following abstract compositional verification principle, where $\mathcal{G}_1$ and $\mathcal{G}_2$ are programs with procedures

[⋆] This work was funded in part by the IST programme of the EC, under the IST-FET-2005-015905 MOBIUS project and under the IST-STREP-27004 S3MS project.

(*i.e.*, components), modelled as *control flow graphs*, and $\uplus$ denotes flow graph composition:

$$\frac{\mathcal{G}_1 \models \sigma \qquad \theta_{I_{\mathcal{G}_1}}(\sigma) \uplus \mathcal{G}_2 \models \phi}{\mathcal{G}_1 \uplus \mathcal{G}_2 \models \phi} \tag{1}$$

Informally, this rule says that to prove that the composition $\mathcal{G}_1 \uplus \mathcal{G}_2$ satisfies property ϕ, it is sufficient to find a "local" property σ of flow graph $\mathcal{G}_1$ (typically a still unavailable component) for which one can verify that: *(i)* σ indeed holds for $\mathcal{G}_1$, and *(ii)* the local property ensures the global property. Task *(i)* is deferred until component $\mathcal{G}_1$ becomes available. Task *(ii)* assumes knowledge of the names of the provided and required methods of $\mathcal{G}_1$ (its so-called *flow graph interface* $I_{\mathcal{G}_1}$), and is achieved by constructing a maximal flow graph for the local property, *i.e.*, $\theta_{I_{\mathcal{G}_1}}(\sigma)$ and by showing that its composition with $\mathcal{G}_2$ satisfies ϕ. In both tasks, the verifications can be performed algorithmically, using finite-state and pushdown automata-based model checking, respectively.

A maximal flow graph *w.r.t.* a property σ is a flow graph that simulates all other flow graphs satisfying property σ. This notion is based on the notion of *maximal model* [9], but in addition takes the set of provided and required methods*, i.e.,* the flow graph interface, into account: a maximal flow graph only simulates flow graphs with the same interface. Our technique requires the local requirement σ to be a structural property, while the global requirement ϕ can be either a structural or a behavioural property. This has the advantage that the approach works for relatively simple program models. All formulae are expressed in the fragment of the modal μ-calculus [14] with boxes and greatest fixed-points only. Recently, we have developed a translation from behavioural properties into structural ones [10]. This allows to apply the compositional verification principle also for local behavioural properties, and thus to (indirectly) reuse the global guarantee as a local assumption for the verification of a larger system. However, in this paper, we do not further discuss this, and we simply assume all local properties to be structural.

We have shown soundness and completeness of the compositional verification principle for a basic program model, only considering sequential control flow. This paper discusses under what conditions the principle can be used with finer, more complex program models. For this, we first present a generic program model, and then explicate the conditions that have to be satisfied by each concrete instantiation. To illustrate the approach, we present two concrete instantiations, one extending the basic program model with exceptional control flow and one with multi-threaded control flow. These finer program models are especially tailored to satisfy the above-mentioned conditions, but are intuitive enough to be useful on their own. In addition, we illustrate how extending the program model allows to express (and verify) more complex program properties.

To support our compositional verification method we have developed a tool set. Originally, this was tailored to the basic program model. The basic version of the tool set has been used to demonstrate utility of the method on an industrial smart card case study [11]. This paper contains an overview of the tool set and describes how various parts of it are adapted to support the new instantiations.

Related Work. The maximal model technique for compositional verification is originally developed by Grumberg and Long [9] for the universal fragment of CTL, and later generalised by Kupferman and Vardi [15] for ACTL*. We have adapted the technique to the fragment of the modal μ-calculus with boxes and greatest fixed-points [11]. Our original program model has been inspired by the one of Besson *et al.* [2], who address the problem of verifying stack invariants of Java programs. The model of Recursive State Machines, proposed by Alur *et al.* [1] is also close to ours, while somewhat finer. However, the authors do not address compositional verification of programs with recursion. Still other models exist for capturing the control flow of applications in Java-like languages, see *e.g.,* [18]. However, because of the specific requirements of our compositional verification technique, we cannot directly reuse these models, and instead rely on our own. Several tools exist for the (non-compositional) verification of behavioural program properties. For example, Moped [13,7] and Alfred [20] encode the behaviour of a program as a pushdown system, that is model checked. In particular, the jMoped variation [23] translates Java bytecode to a pushdown system extended with a set of variables, where instructions are directly mapped to transitions of the system. Also closely related is the two-step extraction technique of Obdržálek [19], where a control flow graph of the program is produced first, and the pushdown system is then generated from this graph. However, neither of these translations addresses multi-threading. Further, existing model checkers for multi-threaded Java (such as Bogor[1] and JavaPathFinder[2]) typically use an implicit program representation that is close to the program itself. Then, abstraction is applied to make verification feasible. In contrast, our program model directly abstracts the program behaviour; without this abstraction a maximal flow graph cannot be constructed.

Overview of the paper. Section 2 describes the generic framework for compositional verification, and shows how our original program model is an instance of this. Sections 3 and 4 describe instantiations with exceptional control flow, and with multi-threaded control flow. Finally, Section 5 draws conclusions and discusses other possible instantiations.

2 A Framework for Compositional Verification

This section presents a method for compositional verification of control flow properties based on a generic program model, identifies sufficient conditions for soundness and completeness of the method, instantiates the generic model to the basic model used in [11], and also outlines the tool set supporting this method.

2.1 Program Model

As the basis for our program model, we use a general notion of specification. Both control flow graph structure and behaviour are defined in terms of such specifications. For a detailed account of the basic definitions, we refer to [11].

[1] See `http://bogor.projects.cis.ksu.edu`

[2] See `http://javapathfinder.sourceforge.net`

Definition 1 (Specification). *A* model *over a set of labels L and a set of atomic propositions A is a structure $\mathcal{M} = (S, L, \to, A, \lambda)$, where S is a set of states, $\to \subseteq S \times L \times S$ a labelled transition relation, and $\lambda\colon S \to \mathcal{P}(A)$ a valuation assigning to each state a set of atomic propositions. A* specification *$\mathcal{S}$ is a pair $(\mathcal{M}, \mathbb{E})$, with $\mathcal{M}$ a model and $\mathbb{E} \subseteq S$ a set of* entry *states.*

The *reachable part* of a specification $\mathcal{S} = (\mathcal{M}, \mathbb{E})$ is defined by $\mathcal{R}(\mathcal{S}) = (\mathcal{M}', \mathbb{E})$, where $\mathcal{M}'$ is obtained from $\mathcal{M}$ by deleting all states and transitions not reachable from $\mathbb{E}$. The *disjoint union* of two specifications is defined by $(\mathcal{M}_1, \mathbb{E}_1) \uplus (\mathcal{M}_2, \mathbb{E}_2) = (\mathcal{M}_1 \uplus \mathcal{M}_2, \mathbb{E}_1 \uplus \mathbb{E}_2)$, where $\mathcal{M}_1 \uplus \mathcal{M}_2 = (S_1 \uplus S_2, L_1 \cup L_2, \{in_i(s) \xrightarrow{a} in_i(s') | s \xrightarrow{a} s' \in \mathcal{M}_i\}, A_1 \cup A_2, \lambda)$, where $\lambda(in_i(s)) = \lambda_i(s)$ and in_i (for $i \in \{1, 2\}$) injects S_i into $S_1 \uplus S_2$. The definition of *simulation* between specifications is standard. Notice that simulation is preserved by disjoint union.

$$\mathcal{S}_1 \leq \mathcal{T}_1 \wedge \mathcal{S}_2 \leq \mathcal{T}_2 \Rightarrow \mathcal{S}_1 \uplus \mathcal{S}_2 \leq \mathcal{T}_1 \uplus \mathcal{T}_2 \tag{2}$$

Let $\mathcal{M}eth$ be an infinite set of method names, and let $\mathcal{C}ontr$ be a possibly infinite set of control values (disjoint from $\mathcal{M}eth$) specific for each instantiation of the model (in the program model with exceptions, for instance, it is a set of exception names). Both sets should be disjoint from any reserved symbols. Every control flow graph comes equipped with an interface, specifying the provided and required methods, and the set of legal control values.

Definition 2 (Flow Graph Interface). *A* flow graph interface *is a triple $I = (I^+, I^-, C)$, where $I^+, I^- \subseteq \mathcal{M}eth$ are finite sets of names of* provided *and* required *methods, and $C \subseteq \mathcal{C}ontr$ is a finite set of control values, respectively. We say I is* closed *if $I^- \subseteq I^+$. The* composition *of two interfaces $I_1 = (I_1^+, I_1^-, C_1)$ and $I_2 = (I_2^+, I_2^-, C_2)$ is defined by $I_1 \cup I_2 = (I_1^+ \cup I_2^+, I_1^- \cup I_2^-, C_1 \cup C_2)$.*

The definition of control flow graph structure, or *flow graphs* for short, is also relativised on the notion of *method specification*, which is specific for each concrete instantiation of the generic program model. We require a method specification to be defined as an instance of the general notion of specification. Given such a definition, one can formally define the notion of *flow graph with interface.*

Definition 3 (Flow Graph). Flow graphs *$\mathcal{G}$* with interface *I, written $\mathcal{G} : I$, are inductively defined by*

– *$(\mathcal{M}_m, \mathbb{E}_m) : (\{m\}, M, C)$ if $(\mathcal{M}_m, \mathbb{E}_m)$ is a method specification for m over M and C,*
– *$\mathcal{G}_1 \uplus \mathcal{G}_2 : I_1 \cup I_2$ if $\mathcal{G}_1 : I_1$ and $\mathcal{G}_2 : I_2$.*

A flow graph $\mathcal{G} : I$ is *closed* if its interface I is closed. We use $\leq_s$ to denote *structural simulation* between flow graphs.

Basic Program Model. The compositional verification principle is originally defined for an instance of the generic definition of flow graph, with $\mathcal{C}ontr$ the empty set. In this basic program model, method flow graphs are defined as follows.

```
class Number {

   public static boolean even(int n){
      if (n == 0)
         return true;
      else
         return odd(n-1);
   }

   public static boolean odd(int n){
      if (n == 0)
         return false;
      else
         return even(n-1);

   }
}
```

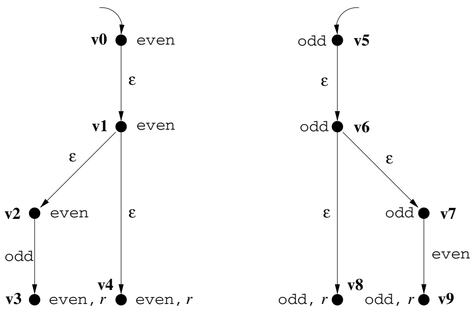

Fig. 1. A simple Java class and its flow graph

Definition 4. (Method Specification) *A* flow graph *for $m \in \mathcal{M}eth$ over a set $M \subseteq \mathcal{M}eth$ is a finite model $\mathcal{M}_m = (V_m, L_m, \rightarrow_m, A_m, \lambda_m)$, with V_m 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., *every node is tagged with its method name). The nodes $v \in V_m$ with $r \in \lambda_m(v)$ are* return points. *A* method specification *for $m \in \mathcal{M}eth$ over M is a pair $(\mathcal{M}_m, \mathbb{E}_m)$ s.t. $\mathcal{M}_m$ is a flow graph for m over M and $\mathbb{E}_m \subseteq V_m$ a non-empty set of* entry points *of m.*

Thus, in this program model, a flow graph $\mathcal{G} : I$ is a model over $I^- \cup \{\varepsilon\}$ and $I^+ \cup \{r\}$.

Example 1. Figure 1 shows a simple Java class and the (simplified) flow graph it induces in the basic program model. The flow graph consists of two method specifications - one for method `even` and one for method `odd`. Entry nodes are depicted as usual through edges without source.

2.2 Model Extraction

The tool set that we developed to support our compositional verification technique contains the *Program Analyser (PA)*, that extracts flow graphs from Java (bytecode) classes. One can always extract a flow graph that over-approximates the actual behaviour as specified by the Java semantics; the precision of the over-approximation depends on the precision of the static analysis used by PA. PA is built on top of the Soot Java Optimization Framework [24]. Soot transforms a bytecode program into Jimple basic blocks. Then, it makes a class hierarchy analysis, producing a safe over-approximation of the application's call graph. For example, if the analysis cannot determine the receiver of a virtual method call, a call edge is generated for every possible method implementation. Further, Soot produces a control flow graph for each method, abstracting away all values. PA transforms these, using information from the call graph, into flow graphs in the format of the program model. Extending PA to the different instantiations amounts to using additional information produced by Soot's different analyses when translating control flow graphs into flow graphs for the program model.

2.3 Flow Graph Behaviour

Next, we define the behaviour of flow graphs. Since the local guarantees must be properties over the flow graph structure, we only have to define the behaviour of closed flow graphs. The behaviour of a flow graph $\mathcal{G}$, denoted $b(\mathcal{G})$, should also be defined as an instance of the general notion of specification. Also on the behavioural level, we instantiate the definition of simulation $\leq_b$: $\mathcal{G}_1 \leq_b \mathcal{G}_2 \Leftrightarrow b(\mathcal{G}_1) \leq b(\mathcal{G}_2)$. For the compositional verification principle to apply for a concrete program model, structural simulation should imply behavioural simulation:

$$\mathcal{G}_1 \leq_s \mathcal{G}_2 \Rightarrow \mathcal{G}_1 \leq_b \mathcal{G}_2 \tag{3}$$

Basic Program Model The behaviour of the basic flow graphs (where $\mathcal{C}ontr = \varnothing$) is defined as follows.

Definition 5. (Behaviour) *Let $\mathcal{G} = (\mathcal{M}, \mathbb{E}) : (I^+, I^-)$ be a closed flow graph such that $\mathcal{M} = (V, L, \rightarrow, A, \lambda)$. The* behaviour *of $\mathcal{G}$ is described by the specification $b(\mathcal{G}) = (\mathcal{M}_b, \mathbb{E}_b)$, where $\mathcal{M}_b = (S_b, L_b, \rightarrow_{bs}, A_b, \lambda_b)$, s.t. $S_b = V \times V^*$, that is, states are* configurations *of control points and stacks, $L_b = \{m_1\ l\ m_2 \mid l \in \{\mathsf{call}, \mathsf{ret}\},\ m_1, m_2 \in I^+\} \cup \{\tau\}$, $A_b = A$, $\lambda_b((v, \sigma)) = \lambda(v)$, and $\rightarrow_{bs}$ is defined as follows:*

[transfer]	$(v, \sigma) \xrightarrow{\tau}_{bs} (v', \sigma)$	if $v \xrightarrow{\varepsilon}_m v'$, $v \models \neg r$
[call]	$(v_1, \sigma) \xrightarrow{m_1\ \mathsf{call}\ m_2}_{bs} (v_2, v_1' \cdot \sigma)$	if $m_1, m_2 \in I^+$, $v_1 \xrightarrow{m_2}_{m_1} v_1'$, $v_1 \models \neg r$, $v_2 \models m_2$, $v_2 \in E$
[return]	$(v_2, v_1 \cdot \sigma) \xrightarrow{m_2\ \mathsf{ret}\ m_1}_{bs} (v_1, \sigma)$	if $m_1, m_2 \in I^+$, $v_2 \models m_2 \wedge r$, $v_1 \models m_1$

The set of entry states $\mathbb{E}_b$ is defined by $\mathbb{E}_b = \mathbb{E} \times \{\epsilon\}$, where ϵ denotes the empty sequence.

Example 2. Consider the flow graph from Example 1. Because of possible unbounded recursion, it induces an infinite-state behaviour. One example execution of the program is represented by the following path from an initial to a final configuration:

$$\begin{aligned}
&(v_0, \epsilon) \xrightarrow{\tau}_{bs} (v_1, \epsilon) \xrightarrow{\tau}_{bs} (v_2, \epsilon) \xrightarrow{\mathsf{even\ call\ odd}}_{bs} (v_5, v_3) \xrightarrow{\tau}_{bs} (v_6, v_3) \xrightarrow{\tau}_{bs} \\
&(v_7, v_3) \xrightarrow{\mathsf{odd\ call\ even}}_{bs} (v_0, v_9 \cdot v_3) \xrightarrow{\tau}_{bs} (v_1, v_9 \cdot v_3) \xrightarrow{\tau}_{bs} \\
&(v_4, v_9 \cdot v_3) \xrightarrow{\mathsf{even\ ret\ odd}}_{bs} (v_9, v_3) \xrightarrow{\mathsf{odd\ ret\ even}}_{bs} (v_3, \epsilon)
\end{aligned}$$

Basic flow graph behaviour can be viewed as the behaviour of a pushdown automaton (PDA). Thus, behavioural properties can be verified using PDA model checking (see [5] for a survey of verification techniques for infinite-state systems). Notice further that for basic flow graphs, structural simulation indeed implies behavioural simulation (thus (3) holds), see [11].

2.4 Properties over Flow Graphs

As property specification language, we use a fragment of the modal μ-calculus [14] with boxes and greatest fixed-points only. A variety of useful safety properties of program control flow structure and behaviour are expressible in this fragment, as illustrated in our earlier work [11]. Let L be a set of labels, A a set of atomic propositions, and V a set of propositional variables.

Definition 6. (Logic) *The formulae of our* 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* $p \in A$, $a \in L$ *and* $X \in V$.

Satisfaction of the logic is defined in terms of the general notion of specification in the standard way [14]. We use $\models_s$ and $\models_b$ to denote instantiation at the structural and behavioural level, respectively: $\mathcal{G} \models_s \phi \Leftrightarrow \mathcal{G} \models \phi$, and $\mathcal{G} \models_b \phi \Leftrightarrow b(\mathcal{G}) \models \phi$.

Example 3. For the flow graph in the basic program model from Example 1, the structural formula $\nu X.\ [\mathsf{even}]\, r \wedge [\mathsf{odd}]\, r \wedge [\varepsilon]\, X$ expresses the property "on every path from a program entry node, the first encountered call edge leads to a return node", in effect specifying that the program is tail-recursive. The behavioural formula $\neg\mathsf{even} \vee \nu X.\ [\mathsf{even\ call\ even}]\, \mathsf{ff} \wedge [\tau]\, X$ expresses the property "in every program execution that starts in method even, the first call is not to method even itself".

Due to the close correspondence between logical satisfaction and simulation, this logic is particularly suited for our compositional verification technique: there exists a mapping χ from finite specifications to formulae, and a mapping (maximal model construction) θ from formulae to finite specifications, such that for any specifications $\mathcal{S}, \mathcal{S}_1$ and finite $\mathcal{S}_2$ (see [11, Ths. 8, 15]):

$$\mathcal{S}_1 \leq \mathcal{S}_2 \Leftrightarrow \mathcal{S}_1 \models \chi(\mathcal{S}_2) \text{ and } \mathcal{S} \models \phi \Leftrightarrow \mathcal{S} \leq \theta(\phi) \tag{4}$$

2.5 Interface Characterisation

As mentioned above, our compositional verification technique is based on the construction of maximal models. However, for a given flow graph property, the maximal model does not necessarily correspond to a legal flow graph structure. Still, if for an interface I we can formulate a *characteristic formula* that precisely defines all legal flow structures with interface I, then we can use this formula to constrain maximal models to legal flow graph structures. Concretely, if σ_I is the characteristic formula for interface I, then the *maximal flow graph* for a property σ is defined as the maximal model (over labels and atomic propositions as induced by I) of the property $\sigma \wedge \sigma_I$. This describes a legal flow graph structure with interface I, simulating all other flow graphs with interface I, satisfying σ. Thus, for any instantiation of the general definition of flow graphs, to be able to apply our compositional verification principle, we need to define a formula σ_I that characterises all flow graphs with interface I, *i.e.*:

$$\mathcal{S} \models \sigma_I \Leftrightarrow \mathcal{R}(\mathcal{S}) : I \tag{5}$$

Basic Program Model. In the basic program model, flow graphs with interface I are models over $I^- \cup \{\varepsilon\}$ and $I^+ \cup \{r\}$ that can be characterised by the following formula ([11, Th. 31]), essentially specifying that every state is labelled by a unique method name that is preserved along edges:

$$\sigma_I = \bigvee_{m \in I^+} \nu X. P_m \wedge [I^-, \varepsilon]\, X \qquad P_m = m \wedge \bigwedge_{m' \in I^+ \setminus \{m\}} \neg m'$$

2.6 Compositional Verification

We can show that compositional verification principle (1) is sound and complete for any instantiation of flow graphs, provided that: *(i)* the notions of method specification and flow graph behaviour are defined as instances of the general notion of specification, (*ii*) structural simulation implies behavioural simulation (property (3)), and *(iii)* flow graphs with interface I can be characterised logically (property (5)). Together with properties (2) and (4), these are sufficient to prove soundness and completeness of the rule (see [11] for a detailed proof). The compositional verification principle applies to the basic program model, as shown in [11].

2.7 A Tool Set for Compositional Verification

In previous work [11], we implemented a tool set to support our compositional verification method in the context of the basic program model presented in Section 2.2. Figure 2 gives a general overview of its architecture.

For each component, we have as input either an implementation (in Java bytecode), or a structural property restricting its possible implementations and an interface specifying the provided and required methods. If we are given the code of the implementation, we use the Program Analyser to extract a flow graph

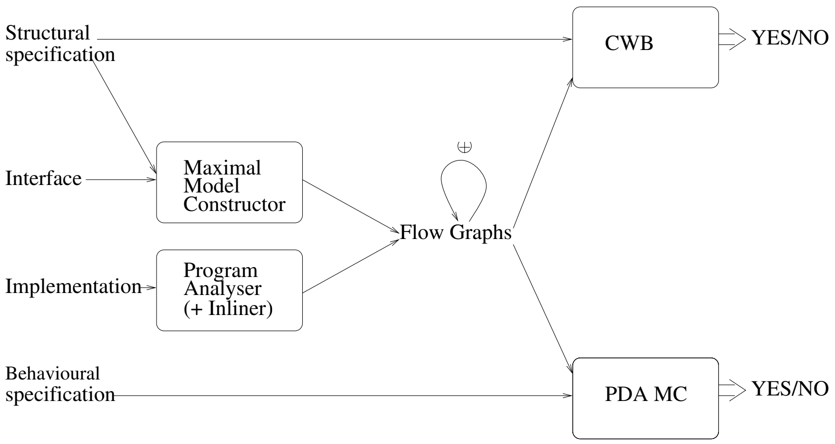

Fig. 2. Tool Set for Compositional Verification

(and if necessary, we use the Inliner to abstract the flow graph to public methods, *i.e.*, methods mentioned in the interface, only [11]). If we are given a structural property, we construct a maximal flow graph as described in Section 2.5 using the Maximal Model Constructor. Composition $\uplus$ of the the resulting flow graphs basically amounts to a concatenation of the textual graph representations. The tool set also implements translations of flow graphs into models which serve as input for different model checkers. In order to check structural properties, we exploit the fact that flow graphs can be viewed as finite Kripke structures, and convert flow graphs to CCS models. Since structural properties are μ–calculus formulae, the verification can then be done using standard model checking tools such as the Concurrency Workbench (CWB) [6]. To verify that a composed system respects a behavioural safety property, we view the behaviour of a flow graph as an infinite state model generated by a Pushdown Automaton (PDA), and apply PDA model checking. We are not aware of an efficient, off-the-shelf model checker for (alternation–free) modal μ–calculus properties of PDAs. We are currently developing one ourselves.

The extensions to the Program Analyser for handling exceptional and multi-threaded control flow are described in the following sections. Extending the Maximal Model Constructor, Inliner and the translation into CCS and PDA models is straightforward, and not discussed further.

The tool set has been evaluated on the PACAP case study [4], an electronic purse developed for smart cards. In PACAP, a smart card may contain one purse applet and several loyalty applets, which interact to exchange information. The case study describes a potential "bad scenario" in terms of an illicit interaction involving the purse applet and the loyalty applets, one of which is malicious. Goal of the verification, presented in detail in [11], is to ensure the absence of such illicit interactions for the given implementations of the purse and loyalties.

3 Instantiation: Exceptional Control Flow

As a first example of how the compositional verification principle can be instantiated to richer program models, we present an instantiation with exceptions. For this, we take $\mathcal{C}ontr$ to be $\mathcal{E}xcp$, an infinite set of exception names, and we define method specifications over $M \subseteq \mathcal{M}eth$ and $E \subseteq \mathcal{E}xcp$.

In a flow graph with exceptions, a control point may be tagged with an exception: the state is said to be exceptional if the current control point is tagged with an exception (*cf.* having an exception at the top of the operand stack [16]). Model extraction from actual bytecode models every instruction that might raise an exception with several transfer edges, one leading to a normal and the others leading to exceptional control points (for all possible exceptions). Explicit throw statements are modelled as internal transfer edges that always lead to an exceptional point. Catch statements are implicit: they are modelled as internal transfer from an exceptional to a normal control point.

At behavioural level, the main difference with the basic model is that the decision in which control point execution resumes after completion of a method call is postponed to the time of return, depending on whether the method call returns normally, or with an exception. Model extraction for a method that may terminate with an exception produces multiple edges labelled with this method, ending in control points tagged with exceptions, in addition to an edge that ends in a normal control point. When a method is called, the set of all possible return points (exceptional and normal) is pushed on the call stack (instead of a single one), so that the appropriate control point can be selected upon return.

Below, we instantiate the compositional verification principle for flow graphs with exceptions in such a way that conditions *(i)*-*(iii)* from Section 2.6 are met. In particular, we define structure and behaviour appropriately. We also discuss how model extraction is adapted, and we give typical example properties that refer to the exceptional structure or behaviour of a flow graph.

3.1 Program Model with Exceptions

As mentioned above, we instantiate $\mathcal{C}ontr$ with $\mathcal{E}xcp$. Interfaces of flow graphs with exceptions are thus of the form (I^+, I^-, E), where $E \subseteq \mathcal{E}xcp$. We use $I^{\mathcal{E}}$ to extract the exception component from the interface.

Method specifications are very similar to method specifications in the basic program model, except that we add exceptions as atomic propositions.

Definition 7. (Method Specification with Exceptions) *A* flow graph with exceptions *for $m \in \mathcal{M}eth$ over sets $M \subseteq \mathcal{M}eth$ and $E \subseteq \mathcal{E}xcp$ is a finite model $\mathcal{M}_m = (V_m, L_m, \rightarrow_m, A_m, \lambda_m)$ with V_m the set of control nodes of m, $L_m = M \cup \{\varepsilon\}$, $A_m = \{m, r\} \cup E$, $m \in \lambda_m(v)$ for all $v \in V_m$, and for all $e, e' \in E$, if $\{e, e'\} \subseteq \lambda_m(v)$ then $e = e'$,* i.e., *each control point is tagged with at most one exception. A* method specification with exceptions *for $m \in \mathcal{M}eth$ over M and E is a specification $(\mathcal{M}_m, \mathbb{E}_m)$ s.t. $\mathcal{M}_m$ is a flow graph with exceptions for m over M and E and $\mathbb{E}_m \subseteq V_M$ a non-empty set of* entry points *of m.*

We use the following abbreviation: $v \models E \Leftrightarrow \exists e \in E.v \models e$. Method specifications with exceptions have to satisfy two *wellformedness* constraints: (1) entry nodes are not exceptional: $\forall v \in \mathbb{E}_m.v \not\models I^{\mathcal{E}}$; and (2) all outgoing edges from exceptional control points are internal transfer edges ending in a normal control point: $\forall v, v' \in V, e \in I^{\mathcal{E}}, l \in L_m.v \models e \wedge v \xrightarrow{l} v' \Rightarrow l = \varepsilon \wedge v' \not\models I^{\mathcal{E}}$. The second constraint is not strictly necessary, but keeps the behaviour of flow graphs clean: catching an exception always results in a normal state in the same method. Throughout, we will assume all method specifications to be wellformed.

3.2 Extracting Flow Graphs with Exceptions from Java Classes

We extended the Program Analyser to handle exceptions. Explicit throw statements give rise to internal transfer edges ending in an appropriately labelled exceptional control point. All other instructions that might raise an exception (such as accessing a reference, which can lead to a NullPointerException) are modelled by a choice: the current control point has multiple outgoing edges labelled ε, one ending in a normal control point and all others ending in appropriate

```
m1();
try { m2();
      try { m3(); }
      catch Exc1 { m4(); }
    }
catch Exc1 { m5(); }
finally { m6(); }
```

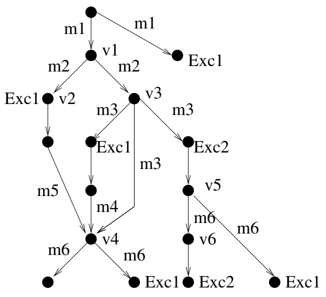

Fig. 3. Example extraction for a try-catch-finally statement

exceptional control points. To model method invocations, edges are labelled either ε, modelling the case that the invocation instruction raises an exception, or with the method name. At most one of the edges labelled with the method name ends in a normal control point, modelling normal termination of the method, all others lead to an exceptional control point, corresponding to exceptional returns from the method. The exceptional control points are either tagged with an exception listed in the method's throw clause, or with a runtime exception that can be thrown (and not caught) in the method. The analysis which exceptions might be returned by a method is transitive *w.r.t.* the call graph.

To illustrate how PA extracts a flow graph from a `try-catch-finally` block, Figure 3 shows an example code fragment[3] together with the corresponding flow graph. We assume that `Exc1` and `Exc2` are the only exceptions; `m1`, `m2` and `m3` and `m6` can throw `Exc1`, while `m3` can also throw `Exc2`. (For presentation purposes, some nodes are named.) A `try-catch` is modelled by branches in the control flow: each instruction in the try-block that could raise an exception has an outgoing edge to an exceptional control point (*e.g.*, the call to `m2` in `v1` can lead to normal point `v3`, or to exception point `v2`). If the exception is handled by one of the catch clauses, the only outgoing edge from this point leads to the control flow of the corresponding clause. For example, in `v2`, the exception is caught by the outer catch clause, leading to a call of `m5`. All edges that correspond to normal termination of the `try-catch` (*i.e.,* termination of the try-block, and termination of all catch-clauses) lead to the same control point, where the flow graph modelling the next instruction starts. If the `try-catch` block is followed by a `finally`-clause, at each possible exit of the `try-catch` block (*e.g.,* nodes `v4` and `v5` in Figure 3), the graph extracted for the `finally` clause is inserted. In case the `try-catch` block ended with an exception, the exception is saved until the end nodes of the graph of the `finally` clause, thus the internal nodes of the `finally` graph are not tagged with this exception. However if an end node of the `finally` graph is normal, an edge is added to rethrow the exception. For example, if the call to `m6` in `v5` ends normally in `v6`, then `Exc2` is re-thrown.

[3] For illustrative purposes, the extraction is described in terms of source code, however the actual implementation works on bytecode.

The end node of a `finally` clause can thus be either normal, tagged with an exception thrown in the `finally` block or with the exception inherited from the `try-catch` block (in case no exception is thrown by the `finally` block itself).

In order to see the results of graph extraction on a realistic piece of software, we analysed a simulation application built on top of the JavaSim library, a tool for building discrete event process-based simulation[4]. We considered 140 types of exceptions, checked as well as unchecked, all subtypes of class `Exception`. The exceptional control flow graph includes 55 methods in 14 classes (approximately 640 lines of code), of which 7 classes belong to the JavaSim library. On a Pentium4 2.2GHz computer with 512MB memory pool, the call graph construction takes 3 minutes, and can be decreased substantially by instrumenting Soot to prevent the analysis of Java API methods. It takes 1,5 seconds to create the control flow graph, which contains 1450 nodes and 1466 edges.

3.3 Flow Graph Behaviour with Exceptions

Modelling the behaviour of flow graphs with exceptions requires a different use of the call stack than in the basic program model. In that model, the return point is determined and pushed on the call stack at the time the method is called. But when modelling exceptional behaviour, it cannot be predicted at call time whether termination will be normal or exceptional. Therefore, the call transition pushes the set of all possible return points on the call stack, and the return transition selects the appropriate one, *i.e.*, with the matching exception (if any). In addition, we introduce transition labels $\mathsf{throw}\, e$ and $\mathsf{catch}\, e$; this makes raising and recovering from exceptions observable for specification purposes.

Definition 8 (Behaviour with Exceptions). *Let $\mathcal{G} = (\mathcal{M}, \mathbb{E}) : I$ be a closed flow graph with exceptions such that $\mathcal{M} = (V, L, \rightarrow, A, \lambda)$. The* behaviour *of $\mathcal{G}$ is described by the specification $b(\mathcal{G}) = (\mathcal{M}_b, \mathbb{E}_b)$, where $\mathcal{M}_b = (S_b, L_b, \rightarrow_{be}, A_b, \lambda_b)$ s.t. $S_b \in V \times (\mathcal{P}(V) \backslash \{\varnothing\})^*$,* i.e., *states are pairs of control points and stacks of non-empty sets of nodes, $L_b = \{m_1\ l\ m_2 \mid l \in \{\mathsf{call}, \mathsf{ret}\},\ m_1, m_2 \in I^+\} \cup \{\tau\} \cup \{l\ e \mid l \in \{\mathsf{throw}, \mathsf{catch}\},\ e \in I^{\mathcal{E}}\}$, $A_b = A$, $\lambda_b((v,\sigma)) = \lambda(v)$ and $\rightarrow_{be}$ is defined as follows:*

$$\begin{array}{rll}
\text{[transfer]} & (v,\sigma) \xrightarrow{\tau}_{be} (v',\sigma) & \text{if } m \in I^+,\ v \xrightarrow{\varepsilon}_m v',\ v \models \neg r,\ v \not\models I^{\mathcal{E}},\ v' \not\models I^{\mathcal{E}} \\
\text{[call]} & (v_1,\sigma) \xrightarrow{m_1\ \mathsf{call}\ m_2}_{be} (v_2, V\cdot\sigma) & \text{if } m_1, m_2 \in I^+,\ v_1 \models \neg r,\ v_1 \not\models I^{\mathcal{E}},\ v_2 \models m_2, \\
 & & \quad v_2 \in \mathbb{E},\ V = \{v \mid v_1 \xrightarrow{m_2}_{m_1} v\},\ V \neq \varnothing \\
\text{[return]} & (v_2, V\cdot\sigma) \xrightarrow{m_2\ \mathsf{ret}\ m_1}_{be} (v_1,\sigma) & \text{if } m_1, m_2 \in I^+,\ v_1 \models m_1,\ v_2 \models m_2 \wedge r, \\
 & & \quad v_1 \in V,\ \forall e \in I^{\mathcal{E}}. v_1 \models e \Leftrightarrow v_2 \models e \\
\text{[throw]} & (v,\sigma) \xrightarrow{\mathsf{throw}\ e}_{be} (v',\sigma) & \text{if } m \in I^+,\ v \xrightarrow{\varepsilon}_m v',\ v \models \neg r,\ v' \models e \\
\text{[catch]} & (v,\sigma) \xrightarrow{\mathsf{catch}\ e}_{be} (v',\sigma) & \text{if } m \in I^+,\ v \xrightarrow{\varepsilon}_m v',\ v \models \neg r \wedge e
\end{array}$$

The set of initial states $\mathbb{E}_b$ is defined by $\mathbb{E}_b = \mathbb{E} \times \{\epsilon\}$.

[4] Available via the JavaSim homepage: `http://javasim.ncl.ac.uk`.

As for the basic model, the behaviour of a flow graph with exceptions is the behaviour of a PDA, and hence PDA model checkers can again be used for verification of behavioural properties. Since there is a close correspondence between flow graph structure and behaviour, structural simulation between flow graphs with exceptions implies their behavioural simulation (thus property (3) holds).

Theorem 1. *Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be flow graphs with exceptions. If $\mathcal{G}_1 \leq_s \mathcal{G}_2$ then $\mathcal{G}_1 \leq_b \mathcal{G}_2$.*

Proof. Let R be a structural simulation between $\mathcal{G}_1$ and $\mathcal{G}_2$. Define relation R_b by (where $|\sigma|$ denotes the length of σ, and $\sigma(i)$ the i^{th} element in σ):

$$(v,\sigma)R_b(v',\sigma') \Leftrightarrow vRv' \wedge |\sigma| = |\sigma'| \wedge \forall i < |\sigma|.\forall w \in \sigma(i).\exists w' \in \sigma'(i).wRw'$$

It is easy to check that R_b is a behavioural simulation between $\mathcal{G}_1$ and $\mathcal{G}_2$. □

3.4 Properties over Flow Graphs with Exceptions

Modelling exceptional control flow of flow graphs not only allows to better approximate their behaviour, it also allows to express and verify properties that are related to exceptions (both at structural and at behavioural level). Typical properties of a flow graph with exceptions $\mathcal{G} : I$ expressible in our logic are:

- Exception $e \in I^{\mathcal{E}}$ is never thrown*:* $\nu X.\neg e \wedge [-]\,X$ (where $[K]\phi$ abbreviates $\bigwedge_{a \in K}[a]\phi$ and '$-$' stands for L). Notice that this property can be expressed both at structural and at behavioural level (but with a slightly different meaning: at the behavioural level, recursion is taken into account, thus certain control points might never be reachable).
- Exception $e \in I^{\mathcal{E}}$ is always caught within the method where it is thrown: $\nu X.(\neg e \vee \neg r) \wedge [-]\,X$ (again, this property can be expressed both at structural and behavioural level).
- After exception $e \in I^{\mathcal{E}}$ is thrown, the first method that can be called is the (state-restoring) method $n \in I^+$: $\nu X.(\neg e \vee \nu Y.\,[M \setminus \{n\}]\,\mathsf{ff} \wedge [\varepsilon]\,Y) \wedge [-]\,X$.

It is natural to handle exceptions locally. Hence, in a compositional verification setting, global behavioural specifications would typically not mention throwing and catching of exceptions; these labels can instead be relabelled into silent τ-transitions.

The tool set has also been extended to translate control flow graphs with exceptions into CCS models. This has been used to produce the CCS model corresponding to the graph extracted for the simulation application described at the end of Section 3.2. Then, we used the Concurrency Workbench to verify various local properties of the application. For instance, we checked whether exceptions are caught locally, i.e., within the method. For the `finalize()` method of JavaSim's `SimulationProcess` class, shown in Figure 4, and a particular exception e, the property `finalize` $\Rightarrow \nu X.(\neg(e \wedge r)) \wedge [-]\,X$ specifies that exceptions of type e are caught locally. The instructions in the `finalize()` method that may raise an exception are the calls to the virtual method `idle()`, the static

```
public void finalize () {
    if (!Terminated) {
           Terminated = true; Passivated = true;
           wakeuptime = SimulationProcess.Never;
           if (!idle()) Scheduler.unschedule(this);
           if (this == SimulationProcess.Current) {
               try { Scheduler.schedule(); }
               catch (SimulationException e) { } }
           SimulationProcess.allProcesses.Remove(this); }}
```

Fig. 4. The `finalize()` method of JavaSim's `SimulationProcess` class

methods `unschedule()`, `schedule()`, `Remove()` and accesses to the fields `Never`, and `Current`. All but one of these instructions raise only the `NullPointerException`: the call to method `schedule()` might raise `NullPointerException` and `SimulationException`, an application-defined exception. Model checking the property succeeded for all exceptions e except for `NullPointerException`, showing that not all exceptions are caught locally.

3.5 Interface Characterisation of Flow Graphs with Exceptions

Given an interface for a flow graph with exceptions I, we can characterise the flow graphs with this interface by the formula σ_I, essentially stating that any initial control point is normal, and after a transition, either the control point is normal again, or we are in an exceptional point, where all outgoing edges are internal transfer edges, leading to a normal control point:

$$\begin{aligned}\sigma_I = {} & \textstyle\bigvee_{m\in I^+}(\nu X.\, P_m \wedge \bigwedge_{e\in I^{\mathcal{E}}} \neg e \wedge \\ & \qquad [I^-, \varepsilon]\,(X \vee (\textstyle\bigwedge_{m\in I^+} [m]\,\mathsf{ff} \wedge P_m \wedge \bigvee_{e\in I^{\mathcal{E}}} P_e \wedge [\varepsilon]X)))\\ P_m = {} & m \wedge \textstyle\bigwedge_{m'\in I^+\setminus\{m\}} \neg m' \qquad\qquad P_e = e \wedge \bigwedge_{e'\in I^{\mathcal{E}}\setminus\{e\}} \neg e'\end{aligned}$$

The following result tells us that σ_I indeed characterises all flow graphs with exceptions with interface I, thus (5) holds.

Theorem 2. *Let I be an interface for flow graphs with exceptions. For any specification $\mathcal{S} = (\mathcal{M}, \mathbb{E})$ over labels $L = I^- \cup \{\varepsilon\}$ and atomic propositions $A = I^+ \cup \{r\} \cup E$ we have (where $\mathcal{R}$ denotes the reachable part of a specification, as defined on page 150): $\mathcal{S} \models_s \sigma_I$ if and only if $\mathcal{R}(\mathcal{S}) : I$.*

Proof. Similar to the proof of Theorem 31 in [11]. □

Because of this result and Theorem 1 the compositional verification principle (1) also applies to flow graphs with exceptions.

4 Instantiation: Multi-threaded Control Flow

As a second example, we instantiate the generic program model with multi-threaded control flow. In this case, the set of control values consists of lock and thread names, *i.e.*, $\mathcal{C}ontr = \mathcal{L}ock \times \mathcal{T}id$, where $\mathcal{L}ock$ and $\mathcal{T}id$ are infinite sets of lock and thread names, respectively. Given an interface I, we use $I^{\mathcal{L}}$ and $I^{\mathcal{T}}$ to extract the legal lock and thread names, respectively.

Our program model supports all basic thread constructs as provided by Java: thread creation, monitors, a wait-notify mechanism, and the possibility to join a thread (*i.e.*, wait for its completion). The behaviour of this instantiation extends the behaviour of the basic program model, by maintaining a configuration for each thread. We assume that (the interleaving behaviours of) programs do not contain data races and thus, by virtue of the Java Memory Model [17], we can assume an interleaving semantics. Notice that the program model described in this section can be easily combined with the program model described above into a single program model with multi-threading and exceptions.

4.1 Program Model with Multi-threading

To define method specifications for multi-threaded programs, we introduce edge labels that correspond to the instructions specific to multi-threading. Following the Java semantics, the body of a method will be executed sequentially, possibly starting new threads, interleaved with other threads. Let $L_{M,L,T}$ abbreviate the set of labels $M \cup \{\varepsilon\} \cup \{c\,l \mid c \in \{\mathsf{lock}, \mathsf{unlock}, \mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\},\ l \in L\} \cup \{\mathsf{spawn}\ t\ \mathsf{with}\ m \mid t \in T, m \in M\} \cup \{\mathsf{join}\ t \mid t \in T\}$.

Definition 9. (Method Specification with Multi-threading) *A* flow graph with multi-threaded control flow *for $m \in \mathcal{M}eth$ over sets $M \subseteq \mathcal{M}eth$, $L \subseteq \mathcal{L}ock$ and $T \subseteq \mathcal{T}id$ is a finite model $\mathcal{M}_m = (V_m, L_{M,L,T}, \rightarrow_m, A_m, \lambda_m)$ with V_m the set of control nodes of m, $A_m = \{m, r\}$, and $m \in \lambda_m(v)$ for all $v \in V_m$. A* method specification with multi-threaded control flow *for $m \in \mathcal{M}eth$ over M, L and T is a specification $(\mathcal{M}_m, \mathbb{E}_m)$ with $\mathcal{M}_m$ a method graph with multi-threaded control flow for m over M, L and T, and $\mathbb{E}_m \subseteq V_m$ a non-empty set of entry points of m.*

4.2 Extracting Flow Graphs from Multi-threaded Java Classes

To extend the Program Analyser to multi-threaded classes, we generate edges with appropriate labels for all (non-deprecated) Java primitives and native methods related to concurrency, with the exception of the timed wait and the interrupt mechanism. For instance, calling the `start` (or `fork`) method on a thread object, is modelled by an edge labelled spawn, while a call to `join` leads to an edge labelled join. Special care is taken for calls to synchronized methods: they are preceded and followed by edges labelled lock and unlock on the appropriate object, *i.e.*, the synchronisation is made explicit.

Special care has to be taken to ensure that the extracted sets of thread and lock names are finite. For threads, a safe over-approximation is to use the declared class name of the thread as thread name in the model. Using a more precise analysis can help to distinguish different threads that are instances of the same class. For locks, abstracting with the class name might under-approximate the program behaviour. To overcome this problem, we require that the program has only a finite number of lock objects with the same class name.

Table 1. Transition rules $\rightarrow_{bm}$ for multi-threaded behaviour

[exec.]	$(\Sigma, \mathsf{L}, \mathsf{W}) \xrightarrow{(t,a)}_{bm} (\Sigma(t{:=}(v', \sigma')), \mathsf{L}, \mathsf{W})$	if $t \notin \mathsf{W}$, $\Sigma(t) \xrightarrow{a}_{bs} (v', \sigma')$
[coord.]	$(\Sigma, \mathsf{L}, \mathsf{W}) \xrightarrow{(t,a)}_{bm} (\Sigma(t{:=}(v', \sigma)), \mathsf{L}', \mathsf{W}')$	if $\Sigma(t) = (v, \sigma), t \notin \mathsf{W}$, $v \xrightarrow{a}_m v'$, $m \in I^+$, $(\mathsf{L}, \mathsf{W}) \xrightarrow{(t,a)}_c (\mathsf{L}', \mathsf{W}')$
[resume]	$(\Sigma, \mathsf{L}, \mathsf{W}) \xrightarrow{(t,\, \mathsf{resume}\ l)}_{bm} (\Sigma, \mathsf{L}', \mathsf{W}')$	if $(t, n, \mathsf{tt}) \in \mathsf{W}(l)$, $\mathsf{L}' = \mathsf{L}(l{:=}(t, n))$, $\mathsf{L}(l) = \perp$, $\mathsf{W}' = \mathsf{W}(l{:=}\mathsf{W}(l) \backslash (t, n, \mathsf{tt}))$
[thr.-ops.]	$(\Sigma, \mathsf{L}, \mathsf{W}) \xrightarrow{(t,a)}_{bm} (\Sigma'(t{:=}((v', \sigma)), \mathsf{L}, \mathsf{W})$	if $\Sigma(t) = (v, \sigma), t \notin \mathsf{W}$, $v \xrightarrow{a}_m v'\ m \in I^+$, $\Sigma \xrightarrow{a}_t \Sigma'$

4.3 Flow Graph Behaviour with Multi-threading

The behaviour specification follows closely the Java Specification [16]. Instead of a single call stack, we maintain a map from thread identifiers to configurations (*i.e.,* control point and call stack). If a thread is not active, it maps to $\perp$. Further, the state space also contains a *lock map* and a *wait map.* The lock map returns for each lock the identity of the thread holding the lock and the lock counter (*i.e.,* how many times the lock is held, necessary to correctly model the reentrant locking behaviour of Java). The wait map returns for each lock the set of threads that are waiting for it, the number of times the thread was holding the lock when it started waiting, and a flag whether the thread has been notified. This ensures that the thread resumes in the exact same state as when it issued a wait, thus making sure a correct number of unlocks is necessary to release the lock. We explicitly require that if a thread is waiting for a lock, its state is active.

We assume execution starts in a special thread called main, and that any closed flow graph contains such a thread. Labels and atomic propositions are paired with thread identifiers. Further, we introduce the atomic proposition $\mathsf{haslock}(t, l)$ to hold in any state where thread t holds lock l.

Definition 10. (Behaviour with Multi-threading) *Let $\mathcal{G} = (\mathcal{M}, \mathbb{E}) : I$ be a closed multi-threaded flow graph such that $\mathcal{M} = (V, L, \rightarrow, A, \lambda)$. The* multi-threaded behaviour *of $\mathcal{G}$ is described by the specification $b(\mathcal{G}) = (\mathcal{M}_b, \mathbb{E}_b)$, where $\mathcal{M}_b = (S_b, L_b, \rightarrow_{bm}, A_b, \lambda_b)$ is defined as follows:*

– $S_b = \{s \in (I^{\mathcal{T}} \rightarrow (V \times V^*)_\perp) \times (I^{\mathcal{L}} \rightarrow (I^{\mathcal{T}} \times \mathbb{N})_\perp) \times (I^{\mathcal{L}} \rightarrow \mathcal{P}(I^{\mathcal{T}} \times \mathbb{N} \times \mathbb{B})) \mid \forall l, t, n, b.(t, n, b) \in \pi_3(s)(l) \Rightarrow \pi_1(s)(t) \neq \perp\}$,
– $L_b = T \times (\{m_1\ c\ m_2 \mid c \in \{\mathsf{call}, \mathsf{ret}\},\ m_1, m_2 \in I^+\} \cup \{\tau\} \cup \{c\, l \mid c \in \{\mathsf{lock}, \mathsf{unlock}, \mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}, \mathsf{resume}\}, l \in I^{\mathcal{L}}\} \cup \{\mathsf{spawn}\ t\ \mathsf{with}\ m \mid t \in I^{\mathcal{T}}, m \in I^+\} \cup \{\mathsf{join}\ t \mid t \in I^{\mathcal{T}}\})$,
– $\rightarrow_{bm}$ *is defined in Table 1*[5] *(using auxiliary rules $\rightarrow_c$ and $\rightarrow_t$ of Table 2) ,*
– $A_b = (T \times A) \cup \{\mathsf{haslock}(t, l) \mid t \in I^{\mathcal{T}}, l \in I^{\mathcal{L}}\}$*, and*
– $\lambda_b(s) = \{(t, p) \mid t \in I^{\mathcal{T}} \wedge \pi_1(s)(t) \neq \perp \wedge p \in \lambda(\pi_1(\pi_1(s)(t)))\} \cup \{\mathsf{haslock}(t, l) \mid \pi_2(s)(l) \neq \perp \wedge \pi_1(\pi_2(s)(l)) = t\}$.

[5] We abbreviate $\exists n, b, l.(t, n, b) \in \mathsf{W}(l)$ as $t \in \mathsf{W}$. We use $f(i{:=}x)$ to denote function update. Further, $\Sigma(i) = (v, \sigma)$ implicitly implies that $\Sigma(i) \neq \perp$.

Table 2. Auxiliary transition rules $\rightarrow_c$ and $\rightarrow_t$

$$\begin{array}{rll}
\text{[lock]} & (\mathsf{L},\mathsf{W}) \xrightarrow{(t,\mathsf{lock}\ l)}_c (\mathsf{L}',\mathsf{W}) & \text{if } \mathsf{L}(l) = \perp,\ \mathsf{L}' = \mathsf{L}(l{:=}(t,1)) \\
\text{[re-lock]} & (\mathsf{L},\mathsf{W}) \xrightarrow{(t,\mathsf{lock}\ l)}_c (\mathsf{L}',\mathsf{W}) & \text{if } \mathsf{L}(l) = (t,n),\ \mathsf{L}' = \mathsf{L}(l{:=}(t,n)) \\
\text{[unlock]} & (\mathsf{L},\mathsf{W}) \xrightarrow{(t,\mathsf{unlock}\ l)}_c (\mathsf{L}',\mathsf{W}) & \text{if } \mathsf{L}' = \mathsf{L}(l{:=}\,(\mathsf{L}(l) = \perp \vee \mathsf{L}(l) = (t,1)\,)? \\
& & \qquad\qquad\qquad \perp: \\
& & \qquad\qquad\qquad (\pi_1(\mathsf{L}(l)), \pi_2(\mathsf{L}(l)) - 1)) \\
\text{[wait]} & (\mathsf{L},\mathsf{W}) \xrightarrow{(t,\mathsf{wait}\ l)}_c (\mathsf{L}',\mathsf{W}') & \text{if } \mathsf{L}(l) = (t,n),\ \mathsf{L}' = \mathsf{L}(l{:=}\perp), \\
& & \quad \mathsf{W}' = \mathsf{W}(l{:=}\mathsf{W}(l) \cup \{(t,n,\mathsf{ff})\}) \\
\text{[notify]} & (\mathsf{L},\mathsf{W}) \xrightarrow{(t,\mathsf{notify}\ l)}_c (\mathsf{L},\mathsf{W}') & \text{if } \mathsf{L}(l) = (t,n), (t',n,\mathsf{ff}) \in \mathsf{W}(l), \\
& & \quad \mathsf{W}' = \mathsf{W}(l{:=}\mathsf{W}(l) \backslash \{(t',n,\mathsf{ff})\} \cup \{(t',n,\mathsf{tt})\}) \\
\text{[notify-cont]} & (\mathsf{L},\mathsf{W}) \xrightarrow{(t,\mathsf{notify}\ l)}_c (\mathsf{L},\mathsf{W}) & \text{if } \mathsf{L}(l) = (t,n), \forall t'.(t',n,\mathsf{ff}) \notin \mathsf{W}(l) \\
\text{[notifyAll]} & (\mathsf{L},\mathsf{W}) \xrightarrow{(t,\mathsf{notifyAll}\ l)}_c (\mathsf{L},\mathsf{W}') & \text{if } \mathsf{L}(l) = (t,n), \\
& & \quad \mathsf{W}' = \mathsf{W}(l{:=}\{(t',n,\mathsf{tt}) \mid (t',n,r) \in \mathsf{W}(l)\})
\end{array}$$

$$\begin{array}{rll}
\text{[spawn]} & \Sigma \xrightarrow{\mathsf{spawn}\ t'\ \mathsf{with}\ m'}_t \Sigma' & \text{if } \Sigma(t') = \perp,\ m' \in I^+,\ v'' \in E,\ v'' \models m',\ \Sigma' = \Sigma(t'{:=}(v'',\epsilon)) \\
\text{[join]} & \Sigma \xrightarrow{\mathsf{join}\ t'}_t \Sigma & \text{if } \Sigma(t') = (v'',\epsilon),\ v'' \models r
\end{array}$$

The set of initial states $\mathbb{E}_b$ *is defined as* $\mathbb{E}_b = \{(\Sigma_I^v, \lambda l.\ \perp, \lambda l.\varnothing) \mid v \in \mathbb{E}\}$ *where* $\Sigma_I^v(\mathsf{main}) = (v, \epsilon, \perp)$ *and* $\Sigma_I^v(t) = \perp$ *for all* $t \in I^{\mathcal{T}}$.

The transition rules should be understood as follows. Rule [exec.] lifts the standard rules for sequential flow graphs ($\rightarrow_{bs}$, Def. 5) to the multi-threaded case. Rule [coord.] models the coordination of threads via locks, *i.e.,* (un)lock, wait, and notify(All): the current thread changes control point if the lock and wait map can be updated appropriately, as defined by the auxiliary transition rules $\rightarrow_c$ (see [12] for more explanation). Rule [thr.-ops.] models creating and joining a thread using the auxiliary transition rules $\rightarrow_t$ (see also [12]). Finally, rule [resume] handles the case where an thread is waiting on an object, has been notified, and now continues execution.

Also in the case of multi-threaded flow graphs, there is a direct correspondence between flow graph structure and behaviour, and thus structural simulation implies behavioural simulation.

Theorem 3. *Let* $\mathcal{G}_1$ *and* $\mathcal{G}_2$ *be flow graphs with multi-threading. If* $\mathcal{G}_1 \leq_s \mathcal{G}_2$ *then* $\mathcal{G}_1 \leq_b \mathcal{G}_2$.

Proof. Let R be a structural simulation between $\mathcal{G}_1$ and $\mathcal{G}_2$. Define

$$\begin{array}{l}
(\Sigma, \mathsf{L}, \mathsf{W}) R_b (\Sigma', \mathsf{L}', \mathsf{W}') \Leftrightarrow \\
(\forall t \in T.\ \mathsf{if}\ \Sigma(t) = (v, \sigma) \\
\qquad\qquad \mathsf{then}\ \Sigma'(t) = (v', \sigma') \wedge vRv' \wedge |\sigma| = |\sigma'| \wedge \forall i.i < |\sigma|.\sigma(i) R \sigma'(i) \\
\qquad\qquad \mathsf{else}\ \Sigma'(t) = \perp) \wedge \mathsf{L} = \mathsf{L}' \wedge \mathsf{W} = \mathsf{W}'
\end{array}$$

It is easy to check that R_b is a behavioural simulation between $\mathcal{G}_1$ and $\mathcal{G}_2$. □

4.4 Properties over Flow Graphs with Multi-threading

The instantiation of the generic flow graph model with multi-threaded control flow allows us to express properties that are related to the multi-threaded character of the flow graph. Given a flow graph $\mathcal{G} : I$ with multi-threaded control flow, typical (behavioural) properties expressible in our logic are:

- Method $m \in I^+$ can only be called by thread t, if t has lock l: $\nu X. \bigwedge t \in I^{\mathcal{T}} (\mathsf{haslock}(t, l) \vee \bigwedge_{m' \in I^+} [(t, m' \ \mathsf{call}\ m)]\, \mathsf{ff}) \wedge [-]\, X$. If method m is the only method accessing some data, this means that data is lock protected.
- Locks are acquired in a particular order, for example lock l_2 can only be acquired by a thread that already has lock l_1: $\nu X. \bigwedge_{t \in I^{\mathcal{T}}} (\mathsf{haslock}(t, l_1) \vee [(t, \mathsf{lock}\ l_2)]\, \mathsf{ff})] \wedge [-]\, X$. This guarantees absence of deadlocks by synchronisation (however, it does not guarantee absence of deadlocks, caused by the wait-notify mechanism, or by joining a non-terminating thread).
- No more than n threads are created in an application. This is an important resource property. Formally, this can be expressed as $\mathit{MaxThr}\ (n)$, inductively defined as follows:

$$\begin{aligned} \mathit{MaxThr}\ (1) &= \nu X_1. \bigwedge_{m \in I^+, t \in I^{\mathcal{T}}} [\mathsf{spawn}\ t\ \mathsf{with}\ m]\, \mathsf{ff} \wedge [-]\, X_1 \\ \mathit{MaxThr}\ (k+1) &= \nu X_{k+1}. \bigwedge_{m \in I^+, t \in I^{\mathcal{T}}} [\mathsf{spawn}\ t\ \mathsf{with}\ m]\, \mathit{MaxThr}\ (k) \wedge [-]\, X_{k+1} \end{aligned}$$

4.5 Interface Characterisation of Flow Graphs with Multi-threading

Given an interface for a flow graph with multi-threaded control flow I, the flow graphs with this interface can be characterised by the formula σ_I, where $L_{M,L,T}$ is as defined on Page 161:

$$\sigma_I = \bigvee_{m \in I^+} (\nu X. P_m \wedge \left[L_{I^-, I^{\mathcal{L}}, I^{\mathcal{T}}} \right] X) \qquad P_m = m \wedge \bigwedge_{m' \in I^+ \setminus \{m\}} \neg m$$

Theorem 4. *Let I be an interface for multi-threaded flow graphs. For any specification $\mathcal{S} = (\mathcal{M}, E)$ over labels $I^- \cup \{\varepsilon\} \cup L_{M,L,T}$ and atomic propositions $A = I^+ \cup \{r\}$ we have : $\mathcal{S} \models_s \sigma_I$ if and only if $\mathcal{R}(\mathcal{S}) : I$.*

Proof. Similar to the proof of Theorem 31 in [11]. □

Thus, the compositional verification principle (1) also applies to flow graphs with multi-threading. However, applying the verification principle poses a problem to model checking, since the verification problem resulting from the second premise, $\theta_I(\phi) \uplus \mathcal{G}_2 \models_b \phi$, is not decidable in general for the case of pushdown systems with multiple stacks. This is a consequence of a basic undecidability result due to Ramalingam [22], which is related to the undecidability of the problem of emptiness of intersection of context-free languages. Hence, every such model checking algorithm must use an under- or over-approximation of the program behaviour. Different approaches have been proposed, see *e.g.,* [3,8,21]. It is future work to study whether and how these solutions can be integrated into our framework.

5 Conclusion

This paper discusses how a previously developed method for compositional verification of control-flow properties of sequential flow graphs with procedures can be adapted to richer program models. We present a generic program model, of which the original program model is an instantiation, and explicate the conditions under which the compositional verification principle is sound and complete. Two other example instantiations of this generic model are presented: with exceptional and with multi-threaded control flow. Also for these particular instantiations, the compositional verification principle holds (noting, however, that in the case of multi-threaded flow graphs we lose decidability due to a general undecidability result for pushdown systems with multiple stacks). The restrictions on the instantiations required to ensure soundness and completeness of the principle are not severe, and the resulting models are intuitive and standard – and can thus be used for other analyses as well. It is future work to study other possibilities to enrich the program model, for example by adding data (from finite domains), or access control information. We are currently adapting the tool set to handle multi-threaded models.

References

1. Alur, R., Benedikt, M., Etessami, K., Godefroid, P., Reps, T., Yannakakis, M.: Analysis of recursive state machines. ACM TOPLAS 27, 786–818 (2005)
2. Besson, F., Jensen, T., Le Métayer, D., Thorn, T.: Model checking security properties of control flow graphs. J. of Computer Security 9(3), 217–250 (2001)
3. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. SIGPLAN Notes 38(1), 62–73 (2003)
4. Bretagne, E., El Marouani, A., Girard, P., Lanet, J.-L.: PACAP purse and loyalty specification. Technical Report V 0.4, Gemplus (2000)
5. Burkart, O., Caucal, D., Moller, F., Steffen, B.: Verification on infinite structures. In: Bergstra, J.A., Ponse, A., Smolka, S.A. (eds.) Handbook of Process Algebra, pp. 545–623. North-Holland, Amsterdam (2000)
6. 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)
7. Esparza, J., Kiefer, S., Schwoon, S.: Abstraction refinement with Craig interpolation and symbolic pushdown systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 489–503. Springer, Heidelberg (2006)
8. Esparza, J., Podelski, A.: Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In: Principles of programming languages (POPL 2000), pp. 1–11. ACM Press, New York (2000)
9. Grumberg, O., Long, D.: Model checking and modular verification. ACM TOPLAS 16(3), 843–871 (1994)
10. Gurov, D., Huisman, M.: Reducing behavioural to structural properties of programs with procedures. Technical Report TRITA-CSC-TCS 2007:3, KTH Royal Institute of Technology, Stockholm (2007)
11. Gurov, D., Huisman, M., Sprenger, C.: Compositional verification of sequential programs with procedures. Information and Computation 206(7), 840–868 (2008)

12. Huisman, M., Aktug, I., Gurov, D.: Flow graph behaviour for multi-threaded applications (2007), `ftp://ftp-sop.inria.fr/everest/Marieke.Huisman/mt.pdf`
13. Kiefer, S., Schwoon, S., Suwimonteerabuth, D.: `http://www.informatik.uni-stuttgart.de/fmi/szs/tools/moped/`
14. Kozen, D.: Results on the propositional μ-calculus. Theoretical Computer Science 27, 333–354 (1983)
15. Kupferman, O., Vardi, M.: An automata-theoretic approach to modular model checking. ACM TOPLAS 22(1), 87–128 (2000)
16. Lindholm, T., Yellin, F.: The Java$^{\text{TM}}$ Virtual Machine Specification, 2nd edn. Sun Microsystems, Inc. (1999)
17. Manson, J., Pugh, W., Adve, S.: The Java memory model. In: Principles of Programming Languages (2005)
18. Méndez, M., Navas, J., Hermenegildo, M.V.: An efficient, parametric fixpoint algorithm for analysis of Java bytecode. In: Huisman, M., Spoto, F. (eds.) Bytecode 2007, pp. 51–66 (2007)
19. Obdržálek, J.: Model checking Java using pushdown systems. In: Proceedings of FTfJP 2002, Malaga, Available as Technical Report NIII-R0204, Computing Science Department, University of Nijmegen (June 2002)
20. Polansky, D.: Implementation of the model checker for pushdown systems and alternation-free mu-calculus. Master's thesis, FI MU Brno (2000)
21. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)
22. Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. ACM TOPLAS 22(2), 416–430 (2000)
23. Suwimonteerabuth, D., Schwoon, S., Esparza, J.: jMoped: A Java bytecode checker based on Moped. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 541–545. Springer, Heidelberg (2005)
24. Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: Soot - a Java Optimization Framework. In: CASCON 1999, pp. 125–135 (1999)