CVPP: A Tool Set for Compositional Verification of Control–Flow Safety Properties

Marieke Huisman[1] and Dilian Gurov[2]*

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

Abstract This paper describes CVPP, a tool set for compositional verification of control–flow safety properties for programs with procedures. The compositional verification principle that underlies CVPP is based on maximal models constructed from component specifications. Maximal models replace the actual components when verifying the whole program, either for the purposes of modularity of verification or due to unavailability of the component implementations at verification time. A characteristic feature of the principle and the tool set is the distinction between program structure and behaviour. While behavioural properties are more abstract and convenient for specification purposes, structural ones are easier to manipulate, in particular when it comes to verification or the construction of maximal models. Therefore, CVPP also contains the means to characterise a given behavioural formula by a set of structural formulae. The paper presents the underlying framework for compositional verification and the components of the tool set. Several verification scenarios are described, as well as wrapper tools that support the automatic execution of such scenarios, providing appropriate pre– and post–processing to interface smoothly with the user and to encapsulate the inner workings of the tool set.

1 Introduction

To enable verification of realistic software, verification techniques have to be compositional and algorithmically decidable. Compositionality ensures that the verification task can be split up in smaller pieces, while algorithmic decidability ensures that verification can be done automatically, without any user interaction. Moreover, for many application domains, compositionality and algorithmic decidability are essential.

For example, in a dynamically reconfigurable distributed system, components can join and leave the system at run–time dynamically. For such an *open system*, appropriate verification techniques are necessary to support safe downloading, *i.e.*, to determine without any user interaction whether a newly arriving component will not corrupt the well–functioning of the global system. These techniques require the *relativisation* of the correctness of the system on the specifications

* Partially funded by the EU FET project FP7–ICT–2009–3 HATS.

and the local correctness of its components. This relativisation can also be used for the purposes of *modularity.* Modular verification is a means of controlling the complexity of verifying large software. It allows an independent local evolution of the implementations of individual modules without affecting the global correctness of the program.

The CVPP tool set is designed to tackle exactly this kind of verification problems by supporting an algorithmic technique for compositional verification. Its focus is on control–flow safety properties of programs with (possibly recursive) procedures. Such properties typically describe sets of allowed sequences of method invocations, and are conveniently expressed in temporal logic. The underlying program model is that of *flow graphs*, abstracting completely from program data to allow efficient algorithmic modular verification. However, the model can be enhanced with exception information or multi–threading. Even though the tool set is developed with compositionality in mind, it can also be used for non–compositional control–flow verification problems of programs with procedures. In particular, it allows to reduce infinite–state verification of behavioural properties to finite–state verification of structural properties.

Abstracting away from all data may seem like a severe restriction, but still many useful properties can be expressed, such as:

- there are no calls to non–atomic methods within atomic transactions;
- in a voting system, candidate selection has to be finished, before the vote can be confirmed;
- a method that changes sensitive data is only called from within a dedicated authentication method, i.e., unauthorized access is not possible;
- in a door access control system, the password has to be checked before the door is unlocked, and it 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.

The present paper describes CVPP, its underlying compositional verification framework, and its implementation. We describe three important verification scenarios: (*i*) open system verification, (*ii*) modular verification, and (*iii*) non–compositional verification. We also discuss the encapsulation of the inner workings of CVPP by means of wrapper tools that automate the various scenarios.

Previous work by the authors on tool support and case studies has been reported in 2004 [15]. The current version of the tool set, discussed in this paper, includes later extensions: (*i*) an inliner to abstract private methods [10], (*ii*) more general program models concerning exceptions, threads and open flow graphs [14,12], and (*iii*) a property translation from behavioural to structural properties [11,12]. The last extension allowed local assumptions to be behavioural, whereas before they had to be structural. Further, we have unified the inputs and outputs to allow interoperability of the individual tools, and have started to work on wrapper tools, automating the verification scenarios.

Related Work Maven is a modular verification tool addressing temporal properties of procedural languages in the context of aspects [8]. A non–compositional verification method based on a program model closely related to ours is presented by Alur and others [3]. It proposes a temporal logic CaRet for nested calls and returns (generalised to a logic for nested words in [1]) that can be used to specify regular properties of local paths within a procedure that skips over calls to other procedures.

Most of the existing work on modular verification of safety properties is based on Hoare logic. Müller was the first to propose a sound modular Hoare–style verification technique for object–oriented languages [17]. A typical verification tool within this line of work is Spec# [4].

Recent work by Alur and Chauhuri proposes a unification of Hoare–style and Manna–Pnueli–style temporal reasoning for procedural programs, presenting proof rules for procedure–modular temporal reasoning [2].

Organisation Sections 2 and 3 sketch the tool set's theoretical background and underlying verification method. Section 4 describes the different tools that make up CVPP, followed by a description of typical verification scenarios in Section 5. Section 6 exemplifies some typical verification tasks when using CVPP. We conclude with possible extensions that would make CVPP applicable to a larger class of problems (without changing the underlying methodology).

2 Program Model and Logic

This section summarises the program model and logic that underlies CVPP. For a more detailed account, the reader is referred to [13].

As mentioned earlier, a characteristic feature of CVPP is the distinction between structural and behavioural properties. Usually, we are interested in properties of the behaviour of a program, while its structure is just a means for accomplishing the desired behaviour. Furthermore, the same behaviour can be produced by several structures. It is thus more natural and more abstract to specify programs with behavioural properties than with structural ones.

However, algorithmic techniques for program analysis and verification are computationally considerably more expensive on the level of program behaviour than on the level of program structure. Program correctness problems are therefore often phrased in terms of the program structure rather than in terms of its behaviour. Furthermore, many behavioural properties have natural structural counterparts, *e.g.*, tail recursion, while other behavioural properties can be characterised through finite sets of structural ones (see Section 3). Therefore, CVPP is set up in such a way that structural properties can be used whenever this is possible and meaningful.

2.1 Model and Logic

Our program model is control–flow based and thus over–approximates actual program behaviour. It defines two different views on programs: a structural and a behavioural one. Both views are instantiations of the general notions of model, defined below. Notice in particular that these instantiations yield a structural and a behavioural version of the logic, and that this enables a uniform treatment of structure and behaviour whenever possible.

Definition 1. (Model) *A* model *is a 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 labelled transition relation, A a set of atomic propositions, $\lambda \colon S \rightarrow \mathcal{P}(A)$ a valuation, assigning to each state s the set of atomic propositions that hold in s. An* initialised model *is a pair $(\mathcal{M}, E)$, with $\mathcal{M}$ a model and $E \subseteq S$ a set of entry states.*

As property specification language we use the fragment of the modal μ-calculus [16] with boxes and greatest fixed-points only. This temporal logic is capable of characterising simulation (*cf.* [13]) and is thus suitable for expressing safety properties. Throughout, we fix a set of labels L, a set of atomic propositions A, and a set of propositional variables V.

Definition 2. (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 on states $(\mathcal{M}, s) \models \phi$ is defined in the standard fashion [16]. For instance, formula $[a]\,\phi$ holds of state s in model $\mathcal{M}$ if ϕ holds in all states accessible from s via an edge labelled a. A model $(\mathcal{M}, E)$ satisfies a formula ϕ, denoted $(\mathcal{M}, E) \models \phi$, if all its entry states E satisfy ϕ. The constant formulae *true* (denoted tt) and *false* (ff) are definable. For convenience, we use $p \Rightarrow \phi$ to abbreviate $\neg p \vee \phi$. We assume that formulae have pair–wise distinct fixed–point binders, and unless stated otherwise, are closed and guarded (*cf.* [22]).

2.2 Control–Flow Structure and Behaviour

Control–Flow Structure We abstract away from all data, therefore program structure is defined as a collection of control–flow graphs (or flow graphs), one for each of the program's methods. Let *Meth* be a countably infinite set of method names. A method graph is an instance of the general notion of model.

Definition 3. (Method graph) *A* method graph *for $m \in Meth$ over a finite set $M \subseteq Meth$ of method names is an initialised 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 $E_m \subseteq V_m$ 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 $\lambda_m \colon V_m \rightarrow \mathcal{P}(A_m)$ is defined 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*.*

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

```
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);

   }}
```

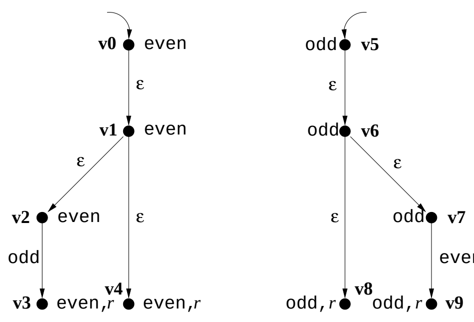

Figure 1. A simple Java class and its flow graph

Flow graph *interfaces* are defined as pairs $I = (I^+, I^-)$, where $I^+, I^- \subseteq$ *Meth* are finite sets of names of *provided* and (externally) *required* methods, respectively[1]. A flow graph $\mathcal{G}$ with interface I is denoted $\mathcal{G} : I$. The flow graph of a program is essentially the (disjoint) union $\uplus$ of its method graphs. Flow graphs can only be composed if their interfaces match. A flow graph is *closed* if $I^- = \varnothing$, *i.e.,* it does not require any external methods. Satisfaction, instantiated to flow graphs, is called structural satisfaction $\models_s$.

Example 2. Consider the flow graph in Example 1. The property "on every path from a program entry node, the first encountered call edge goes to a return node" is formalised by the structural formula $\nu X. [\texttt{even}]\, r \wedge [\texttt{odd}]\, r \wedge [\varepsilon]\, X$, in effect specifying that the program is tail–recursive.

Control–Flow Behaviour Next, we instantiate models on the behavioural level. Transition label τ designates internal transfer of control, m_1 call m_2 designates an invocation of method m_2 by method m_1, and m_2 ret m_1 designates the corresponding return.

Definition 4. (Behaviour) *Let* $\mathcal{G} = (\mathcal{M}, E) : I$ *be a closed flow graph where* $\mathcal{M} = (V, L, \rightarrow, A, \lambda)$. *The* behaviour *of* $\mathcal{G}$ *is defined as the initialised 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 are pairs of control points* v *and stacks* σ *(also called* configurations*),* $L_b = \{m_1\ k\ m_2 \mid k \in \{\mathsf{call}, \mathsf{ret}\},\ m_1, m_2 \in I^+\} \cup \{\tau\}$, $A_b = A$, $\lambda_b((v, \sigma)) = \lambda(v)$, *and* $\rightarrow_b \subseteq S_b \times L_b \times S_b$ *is defined by the rules:*

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

[1] We only require I^- to contain methods that are not provided by I^+. This is different from our earlier work (*e.g.,* [13]), but in line with the tool set implementation.

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

The definition is easily extended to open flow graphs (see [12]). Flow graph behaviour can alternatively be defined via *pushdown automata* (PDA) [13, Def. 34] and approximated with the related notion of pushdown systems (PDS). We exploit this by using PDS model checking for verification of behavioural properties (see [6]). Currently, our tool set relies on the external tool Moped [19]; however, this requires the properties to be translated in LTL.

Example 3. 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 (in the branching structure) from an initial to a final configuration:

$$\begin{array}{l}(v_0, \epsilon) \xrightarrow{\tau}_b (v_1, \epsilon) \xrightarrow{\tau}_b (v_2, \epsilon) \xrightarrow{\textsf{even call odd}}_b (v_5, v_3) \xrightarrow{\tau}_b (v_6, v_3) \xrightarrow{\tau}_b \\ (v_7, v_3) \xrightarrow{\textsf{odd call even}}_b (v_0, v_9 \cdot v_3) \xrightarrow{\tau}_b (v_1, v_9 \cdot v_3) \xrightarrow{\tau}_b \\ (v_4, v_9 \cdot v_3) \xrightarrow{\textsf{even ret odd}}_b (v_9, v_3) \xrightarrow{\textsf{odd ret even}}_b (v_3, \epsilon)\end{array}$$

Also on the behavioural level, we instantiate the definition of satisfaction: we define $\mathcal{G} \models_b \phi$ as $b(\mathcal{G}) \models \phi$. The resulting behavioural logic is powerful enough to express the class of *security policies* defined by finite state security automata [18].

Example 4. For the flow graph from Example 1, the behavioural formula $\texttt{even} \Rightarrow \nu X.\ [\textsf{even call even}]\,\textsf{ff} \wedge [\tau]\,X$ expresses the property "in every program execution starting in method `even`, the first call is not to method `even` itself".

Extensions This section presents the basic program model and logic, considering only normal, sequential control–flow. Extensions with exceptions and with multi–threaded behaviour (with synchronisation on locks) exist [14], and are supported in CVPP. The extension to open flow graphs mentioned above is also supported. In ongoing work we address further extensions to Boolean programs, as well as to richer fragments of the μ–calculus; this is not incorporated in CVPP yet.

3 Framework for Compositional Verification

The compositional verification method underlying our tool set is based on the computation of maximal models from component specifications and the instantiation of components with these models when model checking global system properties. For finite–state systems, this approach was introduced in [9] and since then it has become a standard technique for reducing the verification of correctness of property decompositions to model checking.

Maximal Models for Compositional Verification A model is said to be *maximal* for a given property ϕ, if it satisfies ϕ and simulates (*w.r.t.* a suitable property-preserving simulation relation $\leq$) all models satisfying ϕ. For models in the sense of Definition 1 and formulae in the logic from Definition 2, maximal models exist and are unique up to isomorphism (see [13]). To compute a maximal model for a property ϕ, we present the formula as a modal equation system (see [5]), which is then transformed into a canonical form, the so–called *simulation normal form.* A formula ϕ in simulation normal form can be directly mapped into a (finite) model $\mathcal{M}$ that simulates all models that satisfy ϕ; *i.e.*, for any model $\mathcal{M}'$: $\mathcal{M}' \leq \mathcal{M} \Leftrightarrow \mathcal{M}' \models \phi$. Due to this close connection between simulation and satisfaction, we obtain the following sound and complete verification principle [13]:

> *Compositional verification principle for models*: to show $\mathcal{M}_1 \uplus \mathcal{M}_2 \models \psi$, it suffices to show $\mathcal{M}_1 \models \phi$ (*i.e.,* component $\mathcal{M}_1$ satisfies a suitably chosen *local assumption* ϕ) and $\mathcal{M}_\phi \uplus \mathcal{M}_2 \models \psi$ (*i.e.,* component $\mathcal{M}_2$, when composed with the maximal model $\mathcal{M}_\phi$ for ϕ, satisfies the *global guarantee* ψ).

Completeness of the principle implies that no *false negatives* exist: if $\mathcal{M}_\phi \uplus \mathcal{M}_2 \models \psi$ fails, then there is indeed a model $\mathcal{M}$ such that $\mathcal{M} \models \phi$ but $\mathcal{M} \uplus \mathcal{M}_2 \not\models \psi$.

Adaptation of this principle to flow graphs (as models) and structural and behavioural properties presents us with certain difficulties. Given a structural or behavioural flow graph property ϕ, there is no guarantee that the maximal model of ϕ is a legal flow graph structure or behaviour.

Maximal Flow Graphs from Structural Specifications For structural properties this problem can be solved for a given flow graph interface I, because we can characterise precisely the flow graphs having interface I as models through a structural formula θ_I in our logic. Let $I = \{m_1, m_2\}$ be a closed flow graph interface. A model is a flow graph with this interface exactly when it satisfies the formula $\theta_I = (\nu X.m_1 \wedge [m_1, m_2, \varepsilon]X) \vee (\nu Y.m_2 \wedge [m_1, m_2, \varepsilon]Y)$, which essentially expresses that edges in the flow graph do not cross method boundaries. Then, for every structural formula ϕ, the maximal model of the formula $\phi \wedge \theta_I$ is a flow graph $\mathcal{G}_{\phi,I}$ that simulates structurally all flow graphs with interface I that satisfy ϕ. We term this flow graph the *maximal flow graph* for formula ϕ and interface I, and the compositional verification principle formulated above still applies for flow graphs and structural properties. The above compositional verification principle can then be adapted to structural properties of flow graphs, yielding the following sound and complete compositional verification principle, presented as a proof rule (see [13] for technical details):

$$(\mathsf{struct-comp})\ \frac{\mathcal{G}_1 \models_s \phi \qquad \mathcal{G}_{\phi,I_{\mathcal{G}_1}} \uplus \mathcal{G}_2 \models_s \psi}{\mathcal{G}_1 \uplus \mathcal{G}_2 \models_s \psi}\ \mathcal{G}_1 : I_{\mathcal{G}_1}$$

Maximal Flow Graphs from Behavioural Specifications In the case of behavioural flow graph properties, however, there is no such way to characterise in our logic all models that constitute behaviours of flow graphs with a given interface (intuitively, this is because the logic is not capable of expressing context–free properties). Furthermore, these models are infinite–state and cannot be constructed explicitly; what we actually need is a way to construct the maximal flow graph for a given behavioural formula ϕ and interface I. It turns out, however, that in general there is no such single flow graph, but rather a set of flow graphs having the property that every flow graph satisfying ϕ is simulated by some flow graph in the set. To compute such a set, we have developed a translation from behavioural flow graph properties ϕ to equivalent sets of structural properties $\Pi_I(\phi)$ for a given interface I. The translation is based on a tableau construction that conceptually amounts to symbolic execution of the behavioural formula, collecting structural constraints along the way. By keeping track of the subformulae that have been examined, recursion in the structural constraints is identified and captured by fixed–point formulae (for details see [11]). Combining this translation with maximal flow graph generation for structural properties yields the following sound and complete compositional verification principle for flow graphs and behavioural properties, presented as a proof rule:

$$(\mathsf{beh-comp})\ \frac{\mathcal{G}_1 \models_b \phi \qquad \left\{\mathcal{G}_{\chi, I_{\mathcal{G}_1}} \uplus \mathcal{G}_2 \models_b \psi\right\}_{\chi \in \Pi_{I_{\mathcal{G}_1}}(\phi)}}{\mathcal{G}_1 \uplus \mathcal{G}_2 \models_b \psi}\ \mathcal{G}_1 : I_{\mathcal{G}_1}$$

In addition, we have also developed a "mixed" rule [13], where local structural assumptions are combined with global behavioural guarantees.

The presented proof rules are flexible, so that they allow reasoning about a combination of concrete components (*i.e.*, given through their implementation) and abstract components (*i.e.*, given though their specification), both at the structural and the behavioural levels. Section 5 shows typical verification scenarios, where these proof rules are applied for open system and modular verification. A possible instantiation of this approach is to choose individual methods as components. The proof rules then give rise to a procedure–modular verification technique for temporal properties, see [20].

4 Tool Support for Compositional Verification

This section describes the different internal data formats and tools within the CVPP tool set. It also exemplifies the different input formats used. A high–level overview of CVPP's architecture is shown in Figure 2 (where rounded boxes denote data formats, squared boxes tool components, and dashed lines denote external formats or tools).

As program input format, currently the Java bytecode format is used. Internally, there are three important *data formats:*

– *Model*: the program model representation, containing nodes, edges, a valuation and a set of entry points.
– *Formula*: the property representation. We support behavioural and structural formulae in our logic, both in recursive and in equation system form.
– *Interface:* the interface representation, containing lists of provided and of externally required methods. Interfaces are used as auxiliary information by almost all tool components, and therefore we did not include it explicitly in Figure 2.

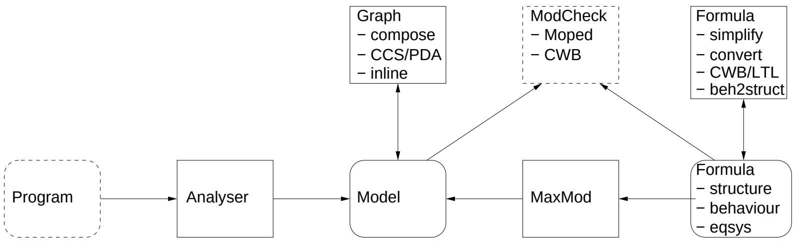

Figure 2. The CVPP tool set architecture

The *components* of the tool set are the following:

- Analyser: from Java classes to flow graphs. Java bytecode classes are abstracted into flow graphs. The tool is build on top of the Soot framework [21].
- Graph: transformations on the program model representations. The main operations supported are flow graph composition, pretty printing in different formats (in particular as CCS process terms and as PDS of the induced behaviour), and inlining of private methods. The use of the latter operation, called Graph Inliner, is briefly explained in Section 5.1 (see also [10]).
- Formula: transformations on the property representations. The main operations supported are the simplification of formulae, the conversion from one property format to another (such as the translation of our logic from recursive to equation system form, needed for maximal model construction), pretty printing as a CWB or LTL formula (as input for Moped), as well as the characterisation of behavioural formulae by structural ones. The latter operation is referred to as Beh2Struct. In addition, we allow properties to be expressed using so–called *patterns*. Patterns provide abbreviations for commonly used specification constructs. They increase readability and make the property more independent of the interface. The Formula component translates patterns into our logic.
- MaxMod: the maximal model construction as described in Section 3. This component uses formulae expressed as equation systems.
- ModCheck: model checking, using external tools: for structural properties we use CWB, the Edinburgh Concurrency Workbench [7], while for behavioural properties we rely on Moped, a PDS model checker for LTL [19].

To conclude this section, we show how the examples from Section 2 are written in CVPP's input formats. Consider again the flow graph from Figure 1. The method graph of method even is written as follows:

```
node 0 meth(even) entry        edge 0 1 eps
node 1 meth(even)              edge 1 2 eps
node 2 meth(even)              edge 1 4 eps
node 3 meth(even) ret          edge 2 3 odd
node 4 meth(even) ret
```

interface for `Number`:	`provided even, odd`
struct. formula Ex. 2:	`nu X.(([even] r) /\ ([odd] r) /\ ([eps] X))`
beh. formula Ex. 4:	`meth(even) => nu X.(([even call even] ff) /\ ([tau] X))`

Figure 3. Examples in CVPP's input format

Figure 3 exemplifies how interfaces and structural and behaviour properties are written in CVPP's input format.

5 Typical Verification Scenarios

Section 3 presented several compositional verification principles; this section describes in detail some typical scenarios supported by CVPP and these verification principles. In addition, we also describe how CVPP can be used for non–compositional verification. This is in particular interesting for behavioural properties: by means of the translation of behavioural properties into structural ones, CVPP provides an effective way to reduce the verification problem for behavioural properties to the computationally simpler problem for structural ones.

5.1 Open System Verification

The most general application of the proof rules presented in Section 3 is to *open system* verification, where some components are given by an implementation (referred to here as concrete components), while others are only given by a specification (abstract components). This can typically happen with dynamically reconfigurable or evolving software, where some components are either not known or simply not statically fixed at verification time.

Thus, verification of a global property of an open system has to be relativised on the local specifications of the abstract components. For instance, if all specifications are behavioural, this is achieved by consecutively applying rule $(\mathsf{beh} - \mathsf{comp})$ on every abstract component. The implementations of the abstract components, once available, are checked against their local specifications.

An additional complexity stems from the detail of information in the concrete components. Often these will contain information about private methods, that are not visible to other components. In contrast, the abstract components and global properties are typically described at the level of the public interface. Therefore, the implementation details in the concrete components are abstracted away, by using the Graph Inliner, to the publicly visible behaviour, before composing the components.

The overall verification task thus divides into two independent tasks, supported by our tool set as follows:

1. *Local correctness*: Check whether the implementation, once available, of every abstract component meets its local specification as described below in Section 5.3.

2. *Global correctness*:
   (a) for every concrete component, from its implementation, extract a flow graph using the Analyser, and use the Graph Inliner to construct its publicly visible behaviour;
   (b) for every abstract component, if its local specification is behavioural, translate the property to an equivalent set of structural ones using Beh-2Struct;
   (c) for every structural property, being either a local specification of an abstract component itself or resulting from step 2(b), compute a maximal flow graph using MaxMod;
   (d) for all instantiations of abstract components by corresponding constructed maximal flow graphs, and instantiations of concrete components by their extracted flow graphs, compose the graphs using Graph to produce a global flow graph of the system, and model check the latter against the global specification as described below in Section 5.3.

5.2 Modular Verification

In the modular software design paradigm the goal is to verify the modules of a software system locally, *i.e.*, independently of each other, and then to combine the local correctness arguments into a global correctness proof of the whole system. In our verification framework, modular verification is simply an instance of the more general case of open system verification described above, with modules as components and where all components are abstract. This eliminates task 2(a) and simplifies conceptually task 2(d).

One can view the notion of module on different levels of granularity. One (rather extreme) case in procedural programming languages is when every procedure itself is considered a module and is equipped with a specification. In this case we obtain *procedure–modular* verification, similar to many Hoare logic based verification approaches. We have recently shown on a case study that it is indeed possible and convenient to reason at this level of granularity about control–flow safety properties of an application [20].

5.3 Non–compositional Verification

The open system and modular verification scenarios above give rise to several non-modular verification tasks. In addition, CVPP also can be used to reason in a fully non-compositional setting. This is in particular useful to reason about behavioural properties. Due to unbounded recursion, verification of behavioural properties for procedural programs is infinite–state, even when all data is abstracted away as in our program model. On the other hand, verification of structural properties is finite–state. Thus, by applying our translation from behavioural to sets of structural properties, one can reduce verification of behavioural properties to a finite number of finite–state verification tasks.

With our tool set, given a Java application and a property specification (either behavioural or structural), perform the following steps:

1. extract the flow graph of the application using the Analyser (and if necessary, use the Graph Inliner to abstract away from implementation details);
2. if the property is structural, cast the flow graph as a CCS term using Graph, and model check the term against the property using the CWB;
3. if the property is behavioural, there are two alternatives: either
   (a) cast the flow graph as a pushdown system using Graph, and model check it against the property using Moped; or
   (b) translate the property to an equivalent set of structural ones using Beh-2Struct, and perform step 2 for each one of these.

Step 3(b) is particularly meaningful in settings where the behavioural specifications are known in advance (such as the security policies of mobile platforms) and are relatively stable; the property translation can then be applied prior to the verification task itself.

5.4 Wrapper Tools for Standard Verification Scenarios

The different scenarios described above require the use of several of the tools of CVPP in a particular pre–defined order. Therefore, to make CVPP easier to use, and to hide away the internal formats and translations within the tool set, *wrapper tools* are being developed that perform the typical verification scenarios automatically. A wrapper implements a pre– and a post–processor that translates input and output of the tool set, and performs the different verification steps automatically. The post–processor appropriately handles feedback from the model checkers: when a structural property is violated, it is indicated where in the program this violation occurs; when a behavioural property is violated the model checking counter example is translated back into a program trace.

The first wrapper tool that we developed is ProMoVer [20]. This automates procedure–modular verification of Java programs annotated with global and method–local specifications. ProMoVer is evaluated on a small but realistic case study: we verified the absence of calls to non–atomic methods within Java Card transactions for a Java Card electronic purse application[2]. In the near future, we plan to develop wrapper tools for the other scenarios.

6 Executing the Verification Scenarios

To illustrate how CVPP is used, this section discusses how parts of the different verification scenarios described in the previous section are applied on concrete examples. For a larger example discussing our experiences with ProMoVer for the verification of the safe use of the Java Card transaction mechanism in an e-commerce application for smart cards, we again refer the reader to [20].

[2] A web–based interface to ProMoVer is available from:
`http://www.csc.kth.se/~siavashs/ProMoVer/promover.php`.

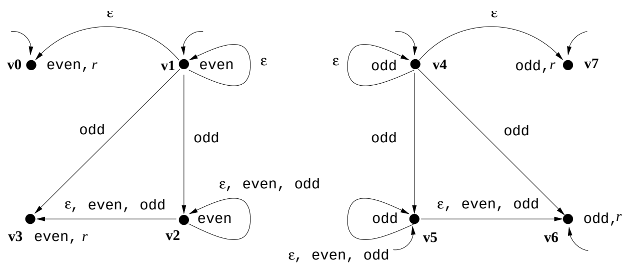

Figure 4. Maximal flow graph for “the first call is not to method `even` itself”

6.1 Generating Maximal Flow Graphs for a Behavioural Property

One important subtask in the compositional verification scenarios discussed in the previous section is the construction of maximal flow graphs from a behavioural specification of a component; see steps 2(b,c) of the open system verification scenario. As explained in Section 3, this is achieved by translating the behavioural property into an equivalent set of structural ones, and by constructing a maximal flow graph for each of the latter.

For example, consider a component specified by an interface where methods `even` and `odd` are provided and no external methods are required, and by the behavioural property “in every program execution starting in method `even`, the first call is not to method `even` itself” formalised in Example 4. Providing this interface and formula to Beh2Struct, and optimising the result with the simplification facility of Formula, we obtain one structural formula: $\mathtt{even} \Rightarrow \nu X.\, [\mathtt{even}]\, \mathsf{ff} \wedge [\epsilon]\, X$. To compute a maximal flow graph, we first apply the conversion facilities of Formula to transform the formula into a modal equation system, which is then passed on, together with the original interface, to MaxMod. The resulting maximal flow graph is shown in Figure 4. Notice that the method graphs for `even` and `odd` are isomorphic, but the graph of method `even` has two entry nodes while the graph of method `odd` has four; as a result, the former restricts the behaviour in that, once called, method `even` can only call method `odd` as a first method call, while the latter makes no restrictions on the behaviour whatsoever. This maximal flow graph can now be substituted for the given component when model checking global system properties.

6.2 Closed System Model Checking of a Behavioural Property

Consider again the component of the previous subsection, described by the interface where methods `even` and `odd` are provided and no external methods are required, and by the behavioural property in Example 4. We want to show that the class `Number` defined in Example 1 is an appropriate implementation of this component. This is an instance of the non-compositional verification scenario in Section 5.3. Thus, using the Analyser, we first extract the flow graph, resulting in the flow graph as in Figure 1. For this application, there is no difference between public and private interface, thus there is no need to use the Graph Inliner.

The property is behavioural, thus we have a choice (*cf.* step 3, Section 5.3). *(a)* We can model check the behavioural property directly. We use Graph to produce the PDS from the flow graph, and Formula to transform the property to an LTL formula. Then Moped is used to verify that class `Number` indeed respects this property. *(b)* As in the previous subsection, we can compute the structural formula that characterises the behavioural formula by using Beh2Struct. We use Graph to pretty print the flow graph as CCS term and Formula to pretty print the formula in CWB's input format. Then CWB is used to verify that class `Number` indeed respects this structural property.

7 Conclusion

CVPP is a tool set for compositional verification of control–flow safety properties of procedural programs. It supports a completely automatic verification method based on maximal models. The underlying general compositional verification principle instantiates to two important verification scenarios, namely open system verification and modular verification. By means of an algorithmic translation of behavioural into structural properties, the tool is also applicable to non–compositional verification, allowing infinite–state PDA model checking to be reduced to standard finite–state model checking. The various scenarios can be supported by wrapper tools, such as ProMoVer, that encapsulate the inner workings of the tool set and provide a smooth interface to the user.

The largest CVPP case study so far is the verification of absence of illicit applet interactions in a smart card application [13,6]. This has been redone with the later extensions of the tool set. It is future work to develop more case studies, similar in size and complexity, but taking advantage of the different wrapper tools. For all three verification scenarios appropriate wrappers will be developed. Further, we plan to provide support for other property specification formalisms, in particular security automata. Also, support for flow graph extraction from source code will be improved, developing a modular and extensible tool. Other extensions concern the program model, where we plan to add data to flow graphs to represent Boolean programs faithfully, and to develop a solution for multi–threaded programs. Finally, we plan to extend the logic to include liveness properties; these become meaningful when the flow graphs model program behaviour faithfully, or at least provide under–approximations of the guaranteed behaviour.

Acknowledgements We thank everybody who contributed to CVPP: Irem Aktug (Analyser), Christoph Sprenger (MaxMod), Siavash Soleimanifard (ProMoVer), and Afshin Amighi (property simplification). We are also indebted to Stefan Schwoon, who extended the input language of Moped to serve our needs.

References

1. R. Alur, M. Arenas, P. Barcelo, K. Etessami, N. Immerman, and L. Libkin. First-order and temporal logics for nested words. In *Logic in Computer Science (LICS '07)*, pages 151–160, Washington, DC, USA, 2007. IEEE Computer Society.

2. R. Alur and S. Chaudhuri. Temporal reasoning for procedural programs. In *Verification, Model Checking, and Abstract Interpretation (VMCAI '10)*, volume 5944 of *LNCS*, pages 45–60. Springer, 2010.
3. R. Alur, K. Etessami, and P. Madhusudan. A temporal logic for nested calls and returns. In *Tools and Algorithms for the Analysis and Construction of Software (TACAS '04)*, volume 2998 of *LNCS*, pages 467–481. Springer, 2004.
4. M. Barnett, K.R.M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS 2004*, volume 3362 of *LNCS*. Springer, 2004.
5. G. Boudol and K. Larsen. Graphical versus logical specifications. *Theoretical Computer Science*, 106:3–20, 1992.
6. G. Chugunov, L.-Å. Fredlund, and D. Gurov. Model checking of multi-applet JavaCard applications. In *Smart Card Research and Advanced Application Conference (CARDIS '02)*, pages 87–95. USENIX Publications, 2002.
7. 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.
8. 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.
9. O. Grumberg and D. Long. Model checking and modular verification. *ACM TOPLAS*, 16(3):843–871, 1994.
10. D. Gurov and M. Huisman. Interface abstraction for compositional verification. In *Software Engineering and Formal Methods (SEFM '05)*, pages 414–423, 2005.
11. 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.
12. D. Gurov and M. Huisman. Reducing behavioural to structural properties of programs with procedures. 2010. Full version, submitted, available upon request.
13. D. Gurov, M. Huisman, and C. Sprenger. Compositional verification of sequential programs with procedures. *Information and Computation*, 206(7):840–868, 2008.
14. M. Huisman, I. Aktug, and D. Gurov. Program models for compositional verification. In *International Conference on Formal Engineering Methods (ICFEM '08)*, volume 5256 of *LNCS*, pages 147–166. Springer, 2008.
15. M. Huisman, D. Gurov, C. Sprenger, and G. Chugunov. Checking absence of illicit applet interactions: a case study. In *Fundamental Approaches to Software Engineering (FASE '04)*, volume 2984 of *LNCS*, pages 84–98. Springer, 2004.
16. D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
17. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer, 2002.
18. F. B. Schneider. Enforceable security policies. *ACM Trans. Infinite Systems Security*, 3(1):30–50, 2000.
19. S. Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technische Universität München, 2002.
20. S. Soleimanifard, D. Gurov, and M. Huisman. Procedure–modular verification of control flow safety properties. In *Workshop on Formal Techniques for Java Programs (FTfJP '10)*, 2010.
21. 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.
22. I. Walukiewicz. Completeness of Kozen's axiomatisation of the propositional mu-calculus. In *Logic in Computer Science (LICS '95)*, pages 14–24. IEEE, 1995.