Algorithmic Verification of Procedural Programs in the Presence of Code Variability

Siavash Soleimanifard and Dilian Gurov

KTH Royal Institute of Technology, Stockholm, Sweden
{siavashs,dilian}@csc.kth.se

Abstract. We present a generic framework for verifying temporal safety properties of procedural programs that are dynamically or statically configured by replacing, adapting, or adding new components. To deal with such a variability of a program, we require programmers to provide local specifications for its variable components, and verify the global properties by replacing these specifications with maximal models. Our framework is a generalization of a previously developed framework that abstracts from all program data. In this work, we capture program data and thus significantly increase the range of properties that can be verified. Our framework is generic by being parametric on the set of observed program events and their semantics. We separate program structure from the behavior it induces to facilitate independent component specification and verification. We provide tool support for an instantiation of our framework to programs written in a procedural language with pointers as the only datatype.

1 Introduction

In modern computing systems code changes frequently. Components evolve rapidly or exist in multiple versions customized for different users, and in open and mobile contexts a system may even automatically reconfigure itself. As a result, systems are no longer developed as monolithic applications; instead they are composed of ready-made off-the-shelf components, and each component may be dynamically replaced by a new one that provides improved or additional functionality. The design and implementation of systems with such static and dynamic *variability* has been attracting considerable attention over the past years. However, there has been less attention to their formal verification. In this paper, we develop a generic framework for the verification of temporal safety properties of such systems.

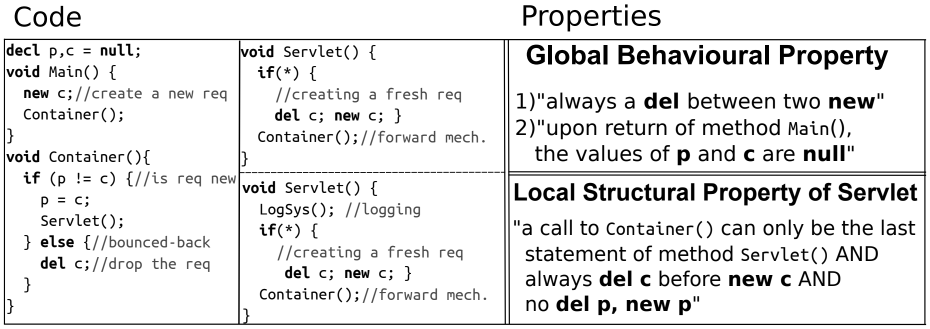
The verification of *variable systems* is challenging because the code of the variable components is either not available at verification time or changes frequently. Therefore, an ideal verification technique for such systems should (i) *localize* the verification of variable components, and (ii) *relativize* the *global* properties of the system on the correctness of its variable components. This can be achieved through a compositional verification scheme where system components are specified *locally* and verified independently, while the correctness of its global properties is inferred from these local specifications. As a result, this allows an independent evolution of the implementations of individual components, only requiring the re-establishment of their local correctness. An algorithmic technique for realization of this verification scheme is to replace the local specifications by so-called *maximal models* [12]. These are most general models satisfying the specifications. Thus, if such models exist, they can replace the specifications of variable components in the verification of the global properties.

The work presented in this paper is the second, final, and conceptually more complicated phase of developing a compositional verification framework for temporal properties of procedural programs with variability exploiting maximal models. In the first phase, we developed a compositional verification technique that separates program structure from its operational semantics (behavior) to allow independent evolution of components [13,15]. The technique abstracts away all program data to achieve algorithmic and practical verification. Such a drastic abstraction, while allowing the verification of certain control flow safety properties [25], significantly reduces the range of properties that can be handled. For instance, properties of sequences of method invocations such as "method m_1 is not called after method m_2 is called" can be verified, but not properties that involve program data, such as "method m_1 is called only if variable `V` is not pointing to `null`". In this work, we generalize this technique to capture program data, and thus bring the usability of our work to a whole new level.

The two main limitations of any verification technique that is based on maximal models are (i) the computationally complex maximal model construction and (ii) the difficulty of producing component specifications. In our previous works, these limitations were softened by full data abstraction. As we show in Section 2, including program data (if done in the straightforward fashion) makes the maximal model construction and property specification impractical: the program models and properties become too detailed and large, maximal model construction becomes unmanageably complex, and the program models become overly specific to one programming language. Our present proposal captures program data without adding extra complexity to the maximal model construction, and keeps the complexity of property specification within practical limits.

We define a novel notion of program structure that is parametric on a set of *actions* that model single instructions of a selected type, and a set of Hoare-style state *assertions* that capture abstractly the effect of a series of statements between consecutive actions. We combine the abstraction provided by assertions with the precision provided by actions to define a uniform control flow graph representation of programs that can be tuned for the verification of the class of properties of interest. The abstraction provided by assertions prevents the local specifications from becoming overly verbose, and allows us to capture program data without adding extra complexity to the maximal model construction. From a wider perspective, by providing Hoare-style assertions and precise ordering of actions these models allow to combine Hoare-style with temporal logic reasoning.

To the extent of our knowledge, our previous framework and consequently the one presented in this paper are the only ones for algorithmic verification of tem-



Code

```
decl p,c = null;
void Main() {
  new c;//create a new req
  Container();
}
void Container(){
  if (p != c) {//is req new
    p = c;
    Servlet();
  } else {//bounced-back
    del c;//drop the req
  }
}
```

```
void Servlet() {
  if(*) {
    //creating a fresh req
    del c; new c; }
  Container();//forward mech.
}
```

```
void Servlet() {
  LogSys(); //logging
  if(*) {
    //creating a fresh req
     del c; new c; }
  Container();//forward mech.
}
```

Properties

Global Behavioural Property

1)"always a **del** between two **new**"
2)"upon return of method `Main()`,
the values of **p** and **c** are **null**"

Local Structural Property of Servlet

"a call to `Container()` can only be the last statement of method `Servlet()` AND always **del c** before **new c** AND no **del p, new p**"

Fig. 1: Web Server Application

poral properties of procedural languages that allow the proofs to be relativized on component specifications. From a technical point of view, the main *contributions* of this paper compared to our previous works are: (i) a novel structural model that combines the precise ordering of selected instructions with abstract representation of the remaining ones, and its operational semantics (a behavioral model), (ii) a proof that the the original maximal model construction can be adapted for the case with data (possibly from infinite domains) with minimal additional cost, (iii) a proof of the correctness of the technique by (non-trivial) re-establishment of our previous results, and (iv) tool support for an instantiation of the framework to programs with pointers as the only datatype. The extended version of this paper including additional examples and proofs can be found in the accompanying report [24].

2 Overview of the Approach

This section provides an overview of our framework by demonstrating its use on an example that mimics the method invocation style of real-life web applications. Although the technique we propose applies to procedural languages in general, we illustrate it here on *Pointer Programs* (PoP), a language with *pointers* as the only datatype [23]. The language supports pointer creation and deletion, assignments and conditional statements, loops, and method-calls with call-by-reference parameter passing. The statement `new x` allocates a fresh chunk of memory and assigns its pointer to variable `x`, while `del x` deletes the memory that `x` is pointing to and assigns `null` to `x` (and all its aliases). The guards for the conditional statements and loops are equality (alias) and inequality checks on variables, and non-deterministic choice, denoted by `*`. Being able to deal with this language is of interest, since it can give rise to infinite state spaces, for two reasons: unbounded stacks of procedure calls, and unbounded pointer creation. Due to space constraints, the formal instantiation of our generic verification framework to the PoP language is delegated to the accompanying report [24].

We use this language to implement a program that mimics the method invocation style of Java enterprise web applications. The execution of such applications starts in method `Container` where based on the current request a *Servlet* is called to prepare the output. As a coding standard [17], servlets should not call each other. Thus if for example servlet A needs to make use of servlet B, it forwards a request to the `Container` that triggers a call to B. We model this so-called forwarding mechanism by explicit invocation of `Container` in servlets.

The program in Figure 1 provides an implementation of a container and two implementations of a single servlet, in which the one at the bottom extends the one at the top by adding a logging facility through calling method `LogSys`. In the code, the variables are pointers to requests. The global variables `p` and `c` point to the previous (last-received) and current requests, respectively. At the beginning of the execution, the request `c` is initialized by `Main` and `Container` is called. In `Container` if the current request is different with the previous one, the current request is stored in `p` and method `Servlet` is called, which non-deterministically generates a fresh request and calls back `Container`. By this, we mimic the call-backs to `Container` (forwarding mechanism) in a real web-application when servlets call each other via the container. `Container` drops (i.e., deletes) the requests that are bounced back to it (when `p = c`) to avoid cycles in the computation. The code of method `LogSys` is not shown here, but we assume that it does not modify the global variables. Here, we consider each method as a component, but in general a component can consist of several methods.

In this example, we assume that the method `Servlet` is the variable part of the program. The structural local specification of method `Servlet` and two behavioral global properties are given in the figure. In the remainder of this section, we explain how to apply to this program the verification technique developed in the later sections, in different variability scenarios.

Verification Technique. In our framework, we divide the verification of variable programs into two independent sub-tasks:

(i) a check that the implementation of each variable component satisfies its local specification, and
(ii) a check that the composition of the local specifications together with the implementations of the non-variable components entails the global property.

By this division we localize the verification of variable components (with sub-task (i)), and relativize the correctness of global properties of the program on the local specifications of its variable components (with sub-task (ii)). Thus, adding or changing the implementation of a variable component does not require the global property to be re-verified, just its local specification (with sub-task (i)). Also notice that, if the local specifications are specified as completely as possible (*i.e.*, are not tailored toward particular global properties), once the local checks of sub-task (i) are performed, the verification of new global properties will not require the re-specification and verification of variable components. In fact, variable components are often implemented and specified as general-purpose libraries that can be used in arbitrary contexts and should thus not be specified toward specific global properties.

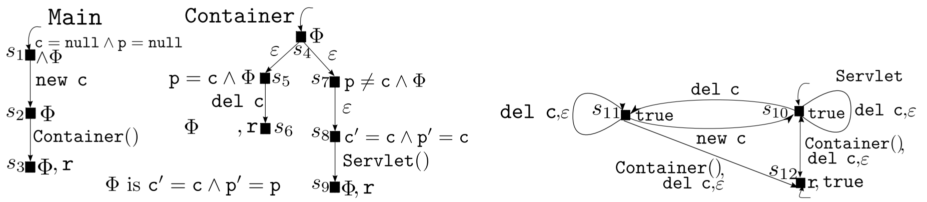
In most variability scenarios, variable systems would be verified once (with sub-tasks (i) and (ii)) before delivering the software to customers, and would be re-verified every time a variable component is modified by performing sub-task (i) on the customer's side. Ideally, sub-task (i) should be performable quickly and thus in isolation from the non-variable part of the system, which is (usually) significantly larger than the modified component. This is difficult to achieve for local specifications that express properties of the execution of programs, *i.e. behavioral* specifications, but is natural for those that express properties of the code (program text) itself, *i.e. structural* specifications. The reason is that the latter can be checked against the component's code rather than the execution of the whole program. For example, a behavioral specification of method `Servlet` would be "`c` points to `null` at any return point of method `Servlet`", which cannot be checked for method `Servlet` in isolation from `Container` and `LogSys`, while the structural specification given in Figure 1 can be checked against `Servlet`'s code, independent from the rest of the program. In practice, these local specifications should be provided by the developers. . This requires the knowledge of the safety requirements of the system.

Let us now mimic some dynamic variability scenarios. First assume that no implementation of `Servlet` is available, for example because it is not implemented yet or should be imported from a third-party library. Still, the incomplete program can be verified from the given structural local property of method `Servlet` and the implementation of methods `Main` and `Container` by performing sub-task (ii). Later, when the implementation of method `Servlet` at the top becomes available, it is only checked against its specification, as in sub-task (i). Now assume that, after a while, the implementation of `Servlet` is updated to the one at the bottom. Again, only the local check of sub-task (i) needs to be performed, this time for the new implementation.

For static variability scenarios, assume that the two implementations of `Servlet` are available and each of them together with `Container` make an application that is delivered to customers based on their needs and budget (as in product families). To verify the global property, the local specification of method `Servlet` is checked for each of the implementations separately (sub-task (i)). Independently, the composition of this local specification with the implementation of `Container` is checked against the global property (sub-task (ii)).

To verify programs in such variability scenarios, we model the structure of non-variable components with *flow graphs*, and convert local specifications of the variable components to *maximal flow graphs*. Here, we present these notions informally and describe how they are used in our verification framework.

Flow Graphs. A flow graph is a finite collection of *method graphs*, each of which represents the control flow structure of a method. Our flow graphs are parametric on the class of program instructions that need to be explicitly represented for the verification of the properties of interest, while using an abstract representation of all other instructions. The rationale is that in temporal reasoning one is usually interested in the ordering of certain events of interest, here called *actions*. The exact ordering of the other events can be abstracted away; only their cumulative



(a) Flow Graphs of `Main` and `Container` (b) Maximal Flow Graph of `Servlet`

Fig. 2

effect needs to be captured. We represent the effect of a series of consecutive events between two actions in a Hoare style, through logical *assertions*. The combination of the precise ordering of actions and abstract representation of data provided by assertions yields a flexible program model that potentially allows to combine Hoare-style with temporal logic reasoning. Here, however, we use these models only for the verification of control flow properties.

In our flow graphs, the actions have parameters and are represented by transition labels, while the assertions are assigned to control nodes[1]. Besides assertions, return nodes are tagged by the atomic proposition `r`. Entry nodes of method graphs represent the beginning of methods.

As an example, Figure 2a shows a flow graph of the code of methods `Main` and `Container`. We want to verify properties talking about order of `new` and `del` statements, e.g., global properties in the figure, thus in this example, actions are `new` and `del`. We add a neutral action ε to simplify the presentation of the flow graphs. Assertions are equality and inequality checks on the variables at the beginning and the end of a block of code between two actions. They express the cumulative effect of condition evaluation and assignments. We use variable names (such as `p` and `c`) and their primed version ($\mathtt{p}'$ and $\mathtt{c}'$) to refer to the values at the beginning and the end of blocks, respectively. For example, state s_8 in the figure represents the assignment statement `p := c` in the code of `Container`.

Maximal Flow Graphs. A maximal flow graph for a specification is a flow graph that represents the structure of *any* code satisfying it. To verify global properties, in our framework the variable components are replaced with maximal flow graphs constructed from their specifications (in sub-task (ii)). By this, we decouple the concrete implementations of variable components from the global correctness reasoning, thus allowing independent evolution of their code. In Section 5, we define formally maximal flow graphs, prove their existence and uniqueness for our specification logic, and provide an algorithm to construct them. Here, we only give an intuitive explanation of their specifics in the present setup.

Local specifications often specify constraints on a small subset of the program variables only, namely the variables whose values should be captured for the ver-

[1] This (maybe non-standard) design choice allows a clear distinction between actions and assertions, which is crucial for our framework.

ification of the class of properties of interest. For example, the specification of method `Servlet` does not specify any constraint on the variables `p` and `c` since their values don't have any effect on the global properties. In such situations, there are (possibly infinitely) many implementations for a component that respect its specification. A maximal flow graph should capture the structure of all these implementations. It is therefore of size exponential in the number of unspecified variables and their values, and is thus infeasible to construct in practice with standard algorithms, e.g., [12,20,13], where data is represented concretely.

In our structural models, however, data is represented symbolically through logical assertions. We use a *semantic entailment* relation on assertions to reduce the size and complexity of the construction of the maximal flow graphs. The idea is that a control node with assertion ϕ can represent any set of nodes that are tagged with assertions entailing ϕ. For example, consider the maximal flow graph constructed from the local specification of `Servlet` shown in Figure 2b. In the graph the assertions (`true`) do not specify any constraints on the variables, so any similar flow graph that for example has $\mathtt{c}' = \mathtt{c}$ or $\mathtt{p}' = \mathtt{p}$ as assertions at its control nodes will be represented by the given maximal flow graph.

Verification. In our framework we support verification of structural and behavioral global properties by performing the sub-tasks (i) and (ii) as follows. (i) The flow graph extracted from the available implementation of `Servlet` is model checked against its local specification. (ii) The maximal flow graph of `Servlet` and the flow graph of `Container` are composed by means of set-theoretic union. This composition can be directly model checked against structural global properties. However, the verification of behavioral global properties requires that a behavioral model is induced from the composition. Intuitively this model (called *flow graph behavior*) should capture all possible runs (executions) of the flow graph. Therefore, it should model the call stack and represent the values of variables at each point of the execution, in which the latter requires the semantics of the transition labels and state assertions. Also, to allow model checking of such models, values should be from finite domains. Then the model can be represented by means of pushdown automata. These models are defined in Section 3.

3 Program Model

We first define an abstract notion of *model* on which our representations of program structure and behavior are based. A model is a *Kripke* structure extended with transition labels and a set of state assertions.

Definition 1 (Model). *A* model *is a tuple* $\mathcal{M} = (S, L, \rightarrow, A, P, \lambda_A, \lambda_P)$ *where* S *is a set of states,* L *a set of labels,* $\rightarrow \subseteq S \times L \times S$ *a labeled transition relation,* A *a finite set of* atomic propositions *(or* atoms*),* P *a finite set of* state assertions*,* $\lambda_A : S \rightarrow 2^A$ *and* $\lambda_P : S \rightarrow P$ *valuations assigning to each state a set of atoms and a state assertion, respectively. An* initialized model $\mathcal{S}$ *is a pair* $(\mathcal{M}, E)$ *with* $\mathcal{M}$ *a model and* $E \subseteq S$ *a set of initial states.*

Models are composed through disjoint union $\uplus$. We assume the set of state assertions P to be equipped with a semantic entailment relation, denoted by $\sqsubseteq$. This relation is used to define simulation preorder, logical satisfaction, and maximal model construction.

In contrast to models without data, the states of models with data are additionally tagged with state assertions. As we shall see, these assertions together with the atomic propositions provide the basis for the symbolic and concrete representation of data, respectively. State assertions are used in structural models to capture how data may change at the states (nodes) of the model, while atomic propositions are used in behavioral models to represent the values of variables at each point of the program execution.

We mentioned that a maximal model is the most general model satisfying a property. The generality relation on models is technically defined w.r.t. a preorder relation called *simulation.* The definition of simulation preorder is parametric on the semantic entailment $\sqsubseteq$.

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

Simulation on two disjoint models $\mathcal{M}_1$ and $\mathcal{M}_2$ is defined, as usual, as simulation on their union. Simulation is extended to initialized models $(\mathcal{M}_1, E_1)$ by defining $(\mathcal{M}_1, E_1) \leqslant (\mathcal{M}_2, E_2)$ if there is a simulation R such that for each $s \in E_1$ there is some $t \in E_2$ with $(s,t) \in R$.

As mentioned earlier, we compose models to verify global properties. The following theorem establishes that simulation is preserved by model composition.

Theorem 1 (Monotonicity). *If $\mathcal{S}_1 \leqslant \mathcal{S}_1'$ and $\mathcal{S}_2 \leqslant \mathcal{S}_2'$ then $\mathcal{S}_1 \uplus \mathcal{S}_2 \leqslant \mathcal{S}_1' \uplus \mathcal{S}_2'$.*

3.1 Flow Graphs

Intuitively, a *flow graph* is a collection of *method graphs*, one for each method of the program, as illustrated in Figure 2a. W.l.o.g., we assume that method names are distinct and taken from a countably infinite set of method names *Meth*. The notion of method graph is an instance of the generic notion of initialized model defined above, with particular sets of assertions P and labels L. Let $\mathcal{A}$ be a set of *actions* with data parameters. The set of flow graph labels is $L = L_{\mathcal{A}} \cup L_{call}$, where $L_{\mathcal{A}} = \{\alpha(a_1, ..., a_n) \mid \alpha \in \mathcal{A}\}$ are action-induced labels and $L_{call} = \{m(a_1, \ldots, a_w) \mid m \in Meth\}$ are labels representing method invocations.

Definition 3 (Method Graph). *A* method graph *for method name $m \in Meth$ over a set $M \subseteq Meth$ of method names is an initialized model $(\mathcal{M}_m, E_m)$ where $\mathcal{M}_m = (S_m, L_m, \rightarrow_m, A_m, P_m, \lambda_{A_m}, \lambda_{P_m})$ is a finite model and $E_m \subseteq S_m$ is a non-empty set of* entry points *of m. S_m is the set of* control nodes *of m, $L_m \subseteq L$, $A_m = \{m, r\}$, $P_m \subseteq P$, $\lambda_{P_m} : S_m \rightarrow P_m$ is a valuation for transition propositions, and $\lambda_{A_m} : S_m \rightarrow \{\{m\}, \{m, r\}\}$ is a valuation for atoms so that* *each node is tagged with its method name, and* return nodes *are additionally tagged with* r.

We sometimes write $s \models m$ to denote $m \in \lambda_A(s)$. Notice that with the above definition, control nodes of flow graphs do not in general correspond to single program points in the actual program's code, but rather to sets of them.

Example 1. The definition of method graphs for PoP programs is an instantiation of the definition above where $\mathcal{A}_{pop}$ is formed from PoP actions and P_m are the PoP assertions. Recall that the set of PoP actions is $\mathcal{A}_{pop} = \{\texttt{del}, \texttt{new}, \varepsilon\}$ where the arities of `new` and `del` are one and of ε is zero. The set of PoP assertions P_{pop} is formed by equality and inequality constraints on the values of variables at the beginning (non-primed variables) and end (primed variables) of the code block that has collapsed into a state. Figure 2a shows a flow graph for the non-variable components (methods `Main` and `Container`) of the PoP program in Figure 1.

Given the definition of PoP assertions above, semantic entailment $\sqsubseteq$ on P_{pop} is defined as logical implication. □

In contrast to the flow graphs defined here, the ones without data do not have state assertions, because all variables and their values are abstracted away.

Every flow graph $\mathcal{G}$ is equipped with an *interface* I, denoted by $\mathcal{G} : I$. A *flow graph interface* consists of a triple $I = (M^+, M^-, \mathit{Modify})$, where $M^+, M^- \subseteq \mathit{Meth}$ are finite sets of *provided* and *required* methods, respectively, and *Modify* is the set of the global variables of the program that are modified in the code of the provided methods. As we shall see, interfaces are needed when constructing maximal flow graphs, which in turn are used for compositional verification.

The definition of *flow graph simulation*, denoted by $\leqslant_s$, is an instantiation of the general notion of simulation on models (see Definition 2) to flow graphs.

3.2 Flow Graph Behavior

Program states $\sigma \in \Sigma$ are defined as usual as mappings from the set of program variables V to their values taken from $\mathcal{D}$. Behavioral transitions conceptually capture the occurrence of actions together with data transformations as specified by assertions. An assertion ϕ is interpreted over pairs of program states, written $(\sigma, \sigma') \models \phi$, and is defined to hold when the closed formula $\phi[\sigma, \sigma']$ is logically valid (here σ, σ' are used as syntactic substitutions for the non-primed and primed variables, respectively). We define *behavioral states* $\langle s, \sigma, \sigma' \rangle$ as consisting of a control node and a pair of program states that satisfies the assertion of the node.

Example 2. PoP programs can create infinitely many pointers. However, at any point of the execution (behavior), only finitely many of them are referenced by program variables. Following [23] we exploit this fact and abstractly represent the infinite pointers of PoP programs by finitely many *equivalence classes*. Two variables are deemed to be equivalent whenever they are pointing to the same memory. Thus, PoP program states are essentially *partitionings* of variables into such equivalence classes. In Example 3 we show how these program states are used to form an execution of the PoP program in Figure 1. □

Next we define flow graph behavior. Behavioral transitions are labeled with "m_1 `call` $m_2(a_1, \ldots, a_w)$" for an invocation of method m_2 by method m_1 with parameters $a_1, \ldots, a_w$, "m_2 `ret` m_1" for the corresponding return from the call, or $\alpha(a_1, ..., a_n) \in L_{\mathcal{A}}$ for the (method-local) transfer of control by action α with parameters $a_1, ..., a_n$. The definition of flow graph behavior is parametric on externally provided (denotational) semantic mappings $[\![\cdot]\!]$ over states and state pairs that specify the (local) effect of actions, calls and returns.

Definition 4 (Flow Graph Behavior). *Let $\mathcal{S} = (\mathcal{M}, E) : (M^+, M^-, \mathit{Modify})$ be a flow graph s.t. $\mathcal{M} = (S, L, \rightarrow, A, P, \lambda_A, \lambda_P)$. The behavior of $\mathcal{S}$ is defined as the initialized model $b(\mathcal{S}) = (\mathcal{M}_b, E_b)$ where $\mathcal{M}_b = (S_b, L_b, \rightarrow_b, A_b, P_b, \lambda_{A_b}, \lambda_{P_b})$, with $S_b \subseteq (S \times \Sigma \times \Sigma) \times (S \times \Sigma)^*$, i.e., states (or configurations) are pairs of behavioral states $\langle s, \sigma, \sigma' \rangle$ and stacks γ over pairs of control nodes and program states, $L_b = L_{\mathcal{A}} \cup \{m_1 \ \boldsymbol{\mathtt{call}} \ l_{m_2} \mid m_1 \in M^+ \wedge l_{m_2} \in L_{call}\} \cup \{m_1 \ \boldsymbol{\mathtt{ret}} \ m_2 \mid m_1, m_2 \in M^+\}$, $A_b = A \cup (\Sigma \times \Sigma)$, $P_b = \{\boldsymbol{\mathtt{tt}}\}$, $\lambda_{A_b}(\langle s, \sigma, \sigma' \rangle, \gamma) = \lambda_A(s) \cup \{(\sigma, \sigma')\}$, and $\rightarrow_b \subseteq S_b \times L_b \times S_b$ is defined by the following transition rules:*

$$
\begin{array}{lll}
[\alpha] & (\langle s_1, \sigma_1, \sigma_1' \rangle, \gamma) \xrightarrow{\alpha(\sigma'(a_1), \ldots, \sigma'(a_n))} (\langle s_2, \sigma_2, \sigma_2' \rangle, \gamma) & \textbf{\textit{if}} \ s_1 \xrightarrow{\alpha(a_1, \ldots, a_n)} s_2 \wedge \\
 & & (\sigma_1, \sigma_1') \models \lambda_P(s_1) \wedge \\
 & & (\sigma_2, \sigma_2') \models \lambda_P(s_2) \wedge \\
 & & \sigma_2 = [\![\alpha]\!] \sigma_1'
\end{array}
$$

$$
\begin{array}{lll}
[\mathit{call}] & (\langle s_1, \sigma_1, \sigma_1' \rangle, \gamma) & \textbf{\textit{if}} \ s_1 \xrightarrow{m'(a_1, \ldots, a_w)} s_2 \ \wedge \ s \models m' \wedge \\
 & \xrightarrow{m \ \mathtt{call} \ m'(\sigma'(a_1), \ldots, \sigma'(a_w))} (\langle s, \sigma, \sigma' \rangle, \langle s_2, \sigma_1' \rangle \cdot \gamma) & s_1, s_2 \models m \ \wedge \ m, m' \in M^+ \wedge \\
 & & (\sigma_1, \sigma_1') \models \lambda_P(s_1) \wedge \\
 & & (\sigma, \sigma') \models \lambda_P(s) \wedge \\
 & & \sigma_2 = [\![\mathit{call}]\!] \sigma_1'
\end{array}
$$

$$
\begin{array}{lll}
[\mathit{ret}] & (\langle s_1, \sigma_1, \sigma_1' \rangle, \langle s_2, \sigma_2 \rangle \cdot \gamma) \xrightarrow{m \ \mathit{ret} \ m'} (\langle s_2, \sigma_3, \sigma_3' \rangle, \gamma) & \textbf{\textit{if}} \ s_1 \models r \ \wedge \ m, m' \in M^+ \wedge \\
 & & s_2 \models m' \ \wedge \ (\sigma_1, \sigma_1') \models \lambda_P(s_1) \wedge \\
 & & s_1 \models m \ \wedge \ (\sigma_3, \sigma_3') \models \lambda_P(s_2) \wedge \\
 & & \sigma_3 = [\![\mathit{ret}]\!](\sigma_1', \sigma_2)
\end{array}
$$

The initial configurations are $E_b = \{(\langle s, \sigma_0, \sigma_0' \rangle, \epsilon) \mid s \in E \wedge (\sigma_0, \sigma_0') \models \lambda_P(s)\}$, where σ_0 and ϵ denote the initial program state and the empty stack, respectively.

In the behavioral models variables are explicitly assigned to values and therefore the set of assertions P_b should be empty. However, to be faithful to Definition 1, we use the (dummy) value `tt` which we assign to all behavioral states. It should further be noted that if $\mathcal{D}$ is finite, flow graph behavior can also be defined by means of *pushdown automata*, as in [13].

Example 3. The above definition is instantiated to PoP programs through the denotational semantics of PoP transition labels: PoP actions, `call` and `ret`. Due to space limitation, here we only provide an intuitive explanation of the semantics and delegate the formal definitions to the accompanying technical report [24]. Intuitively, the semantics of ε is the identity function, `del(`v`)` moves

v and all of its aliases to the equivalence class of `null`, `new(`v`)` maps v to a fresh equivalence class, `call` initializes a program state for the called method, and `ret` recomputes the equivalence classes of variables upon a return from a call. Consider the composition of flow graphs shown in figures 2a and 2b. An example run through this flow graph is shown below[2]. In the run, the boxes represent the equivalence classes of variables, where the left box always represents the class of `null`, e.g., $\boxed{\mathtt{p}}\boxed{\mathtt{c}}$ shows that variables `p` is in the equivalence class of `null` and `c` is in a different one.

$$\begin{aligned}
&(\langle s_1, \boxed{\mathtt{p,c}}\boxed{\,}, \boxed{\mathtt{p,c}}\boxed{\,}\rangle, \epsilon) \xrightarrow{\texttt{new c}} (\langle s_2, \boxed{\mathtt{p}}\boxed{\mathtt{c}}, \boxed{\mathtt{p}}\boxed{\mathtt{c}}\rangle, \epsilon) \xrightarrow{\texttt{Main call Container()}} (\langle s_4, \boxed{\mathtt{p}}\boxed{\mathtt{c}}, \boxed{\mathtt{p}}\boxed{\mathtt{c}}\rangle, \langle s_3, \boxed{\mathtt{p}}\boxed{\mathtt{c}}\rangle)\\
&\xrightarrow{\varepsilon} (\langle s_7, \boxed{\mathtt{p}}\boxed{\mathtt{c}}, \boxed{\mathtt{p}}\boxed{\mathtt{c}}\rangle, \langle s_3, \boxed{\mathtt{p}}\boxed{\mathtt{c}}\rangle) \xrightarrow{\varepsilon} (\langle s_8, \boxed{\mathtt{p}}\boxed{\mathtt{c}}, \boxed{\,}\boxed{\mathtt{p,c}}\rangle, \langle s_3, \boxed{\mathtt{p}}\boxed{\mathtt{c}}\rangle)\\
&\xrightarrow{\texttt{Container call Servlet()}} (\langle s_{10}, \boxed{\,}\boxed{\mathtt{p,c}}, \boxed{\mathtt{p,c}}\boxed{\,}\rangle, \langle s_9, \boxed{\,}\boxed{\mathtt{p,c}}\rangle, \langle s_3, \boxed{\mathtt{p}}\boxed{\mathtt{c}}\rangle)\\
&\xrightarrow{\varepsilon} (\langle s_{12}, \boxed{\mathtt{p,c}}\boxed{\,}, \boxed{\mathtt{p,c}}\boxed{\,}\rangle, \langle s_9, \boxed{\,}\boxed{\mathtt{p,c}}\rangle, \langle s_3, \boxed{\mathtt{p}}\boxed{\mathtt{c}}\rangle) \xrightarrow{\texttt{Servlet ret Container}} (\langle s_9, \boxed{\mathtt{p,c}}\boxed{\,}, \boxed{\mathtt{p,c}}\boxed{\,}\rangle, \langle s_3, \boxed{\mathtt{p}}\boxed{\mathtt{c}}\rangle)\\
&\xrightarrow{\texttt{Container ret Main}} (\langle s_3, \boxed{\mathtt{p,c}}\boxed{\,}, \boxed{\mathtt{p,c}}\boxed{\,}\rangle, \epsilon)
\end{aligned}$$

Observe how assertions and transitions change the equivalence classes of variables, and states are pushed to and popped from the stack. E.g., the first transitions, `new c`, changes the equivalence class of `c` from `null` to a fresh one; the assertion of state s_8 moves `p` to the equivalence class of `c`; and the second transition pushes $\langle s_3, \boxed{\mathtt{p}}\boxed{\mathtt{c}}\rangle$ to the stack, that is popped by the last transition. □

In contrast to the above definition of behavior, the one without data does not have program states Σ, and the only action is ε. Thus, at calls control nodes are simply pushed to the stack and these are popped at returns. Also the set of atomic propositions A_b is equal to A, only consisting of method names and `r`.

Again, we instantiate the general definition of simulation (Definition 2) to flow graph behavior, and denote it by $\leqslant_b$. A result that we later exploit for compositional verification is that if two flow graphs are related by structural simulation, then their behaviors are related by behavioral simulation.

Theorem 2 (Simulation Correspondence). *For flow graphs A and B, if $A \leqslant_s B$ then $b(A) \leqslant_b b(B)$.*

4 Logic

As a property specification language we use the safety fragment of Modal Equation Systems [21], that is without diamond modalities. This logic is equal in expressive power to the safety fragment of the modal μ-calculus [19]. Here, we employ the former logic for technical reasons that will become clear later, but a user is free to use either. The translation of μ-calculus to simulation logic defined in Definition 6 below is based on Bekič's principle described in [8,6]. The translation in the other direction is straightforward and done simply by replacing each fixed point by an equation.

[2] This example is simplified for the presentation in this paper. For complete examples please see [24].

Following Larsen [21], we define the syntax and semantics of the specification language in two steps: first we define a basic modal logic that is parametrized on a set of labels L, state assertions P, and atoms A, and then we add recursion by means of equation systems in Definition 6. *Basic simulation logic* is a variant of Hennessy-Milner logic [14] without diamond modalities.

Definition 5 (Basic Simulation Logic: Syntax). *The formulas of* basic simulation logic *are inductively defined by:*

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

where $a \in A$, $p \in P$, $l \in L$, and X ranges over a set of propositional variables $\mathcal{V}$. Formulas of the shape a, $\neg a$, p, and $\neg p$ are called atomic formulas*.*

The semantics of a formula ϕ of basic simulation logic over L, P, and A is defined relative to a model M and an environment $\rho : \mathcal{V} \to 2^S$ as an extension of the standard definition (see [26]) with the following additional clauses.

$$\|p\|\rho \overset{\text{def}}{=} \{s \in S \mid \lambda_P(s) \sqsubseteq p\} \qquad \text{and} \qquad \|\neg p\|\rho \overset{\text{def}}{=} S \setminus \|p\|\rho$$

Definition 6 (Modal Equation System). *A* modal equation system $\varPi = \{X_i = \varPhi_i \mid i \in J\}$ *over L and A for a set of indexes J is a finite set of* defining equations *such that the variables X_i are pairwise distinct and each $\varPhi_i$ is a formula of basic simulation logic over L, P, and A. The set of variables occurring in $\varPi$ is partitioned into the set of* bound *variables, defined by* $\mathtt{bv}(\varPi) = \{X_i \mid i \in J\}$*, and the set of* free *variables* $\mathtt{fv}(\varPi)$*.*

The semantics of a closed modal equation system $\|[\varPi]\|\rho$ is defined as its greatest fixed point. We use n-ary versions of conjunction and disjunction, setting $\bigwedge \emptyset = \mathtt{tt}$ (*true*) and $\bigvee \emptyset = \mathtt{ff}$ (*false*). We use *Labels*(X) and *Atoms*(X) to refer to the set of labels and atoms of the defining equation for X, respectively.

Finally, using the definitions of basic simulation logic and modal equation systems, the formulas of *simulation logic* are defined by $\varPhi[\varPi]$ over L, P, and A, where $\varPhi$ is a formula of basic simulation logic and $\varPi$ is a modal equation system. The semantics of $\varPhi[\varPi]$ w.r.t. model M and environment ρ is defined by $\|\varPhi[\varPi]\|\rho \overset{\text{def}}{=} \|\varPhi\|\rho[\|\varPi\|\rho]$. We say that a state s of a model M satisfies $\varPhi[\varPi]$, written $(M, s) \models \varPhi[\varPi]$, if $s \in \|\varPhi[\varPi]\|\rho$ for all ρ. For initialized model (M, E) we define $(M, E) \models \varPhi[\varPi]$ if $(M, s) \models \varPhi[\varPi]$ for all $s \in E$.

Simulation logic is capable of expressing safety properties of sequences of observed actions, calls and returns. We use two instantiations of this logic to represent structural and behavioral properties. Structural logic expresses properties of flow graphs (Definition 3), therefore it is instantiated by $a \in A$, $p \in P$, and $l \in L$. Behavioral logic, however, expresses properties of flow graph behaviors (Definition 4), therefore it is instantiated by $a \in A_b$, $p \in P_b$, and $l \in L_b$.

Example 4. The structural local property "`Container` can only be called as the last statement of the method `Servlet`" in Figure 1 is specified by the structural formula $X[\varPi]$, where $\varPi$ is

$$X = [\mathtt{Container()}]\mathtt{r} \wedge \bigwedge_{l \in L_{\mathtt{Servlet}} \setminus \mathtt{Container()}} [l]X$$

The second behavioral global property in Figure 1 is specified by the behavioral formula $X[\Pi]$, where Π is $X = ((\texttt{Main} \wedge \texttt{r}) \Rightarrow (\texttt{p} = \texttt{null} \wedge \texttt{c} = \texttt{null})) \wedge \bigwedge_{l \in L_b} [l]X$.

5 Maximal Models and Flow Graphs

To construct maximal models, we generalize our previous algorithm for models without program data [13], following closely the treatment there. We therefore only sketch our construction here, and refer the reader to [13] for the details. Our construction algorithm is defined on the general notion of model (Definition 1).

5.1 Maximal Model Construction

We define two auxiliary functions θ and χ which form a *Galois connection* between finite models and formulas in simulation logic. Function χ translates a *finite* model into a formula, while θ translates a formula into a (finite) model. Both functions are defined on formulas in a so-called *simulation normal form* (SNF). In this section, we define SNF and show that every formula of simulation logic has an SNF representation and provide an algorithm to convert a formula to its SNF. The construction of maximal models basically consists of translating a given formula into SNF and applying function θ on the result.

Definition 7 (χ). *Function χ maps a finite initialized model $(\mathcal{M}, E)$ into its* characteristic formula $\chi(\mathcal{M}, E) = \phi_E[\Pi_\mathcal{M}]$*, where $\phi_E = \bigvee_{s \in E} X_s$, and $\Pi_\mathcal{M}$ is defined by the equations:*

$$X_s = \bigwedge_{l \in L} [l] \bigvee_{s \xrightarrow{l} t} X_t \wedge \bigwedge_{a \in \lambda_A(s)} a \wedge \bigwedge_{b \notin \lambda_A(s)} \neg b \wedge \lambda_P(s)$$

Example 5. Function χ maps the flow graph of method `Main` to its characteristic formula $(X_1)[\Pi_{\texttt{Main}}]$, where $\Pi_{\texttt{Main}}$ is the modal equation system

$$\begin{aligned}
X_1 &= \textstyle\bigwedge_{l \in L \setminus \{\texttt{new c}\}} [l]\texttt{ff} \wedge [\texttt{new c}]X_2 \wedge \neg r \wedge \texttt{Main} \wedge \Phi \\
X_2 &= \textstyle\bigwedge_{l \in L \setminus \{\texttt{Container()}\}} [l]\texttt{ff} \wedge [\texttt{Container()}]X_3 \wedge \neg r \wedge \texttt{Main} \wedge \Phi \\
X_3 &= \textstyle\bigwedge_{l \in L} [l]\texttt{ff} \wedge r \wedge \texttt{Main} \wedge \Phi
\end{aligned}$$

and where L is the set of structural labels. Recall that $\texttt{ff} = \bigvee \emptyset$. □

The next result shows that function χ precisely translates an initialized model to a formula. This is a variation of an earlier result by Larsen [21].

Theorem 3. *Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be two initialized models and let $\mathcal{S}_2$ be finite. Then $\mathcal{S}_1 \leqslant \mathcal{S}_2$ if and only if $\mathcal{S}_1 \models \mathcal{X}(\mathcal{S}_2)$.*

Definition 8 (Simulation Normal Form). *A formula $\phi[\Pi]$ of simulation logic over L, A, and P is in* simulation normal form (SNF) *if ϕ has the form*

$\bigvee \mathcal{Z}$ *for some finite set* $\mathcal{Z} \subseteq \mathtt{bv}(\Pi)$ *and all equations of* Π *have the following* state normal form

$$X = \bigwedge_{l\in L} [l] \bigvee \mathcal{Y}_{X,l} \wedge \bigwedge_{a\in B_X} a \wedge \bigwedge_{b\notin A\setminus B_X} \neg b \wedge p \tag{1}$$

where each $\mathcal{Y}_{X,l} \subseteq \mathtt{bv}(\Pi)$ *is a finite set of variables,* $B_X \subseteq A$ *is a set of atomic propositions, and* $p \in P$ *is a state assertion.*

Example 6. The property shown in Example 5 is in simulation normal form. □

To translate simulation logic formulas into SNF we generalize the algorithm of [13] that works as follows. For a given set of atoms A, labels L, and a formula $\phi[\Pi]$, it saturates each equation of Π by conjoining its missing labels as $\bigwedge_{l\in L \wedge l \notin \mathit{Labels}(X)} [l]\ \mathtt{tt}$, and atoms as $\bigwedge_{a\in A\wedge a\notin \mathit{Atoms}(X)} (a \vee \neg a)$, and then transforms the resulting formula to SNF by introducing new equations for disjunctions of formulas not guarded by any box. Our adaptation of this algorithm to formulas $\phi[\Pi]$ of Definition 6 proceeds in two steps. First, we apply the above algorithm to $\phi[\Pi]$, simply carrying over the assertions of the equations. In the second step, we conjoin the top element of the lattice of P to the resulting equations that do not have any assertions. In this way we simplify the saturation of assertions, that would otherwise be very inefficient or even impossible when the set of variables and their values is large or infinite.

Theorem 4. *Every formula of simulation logic has an equivalent one in SNF.*

Definition 9 (θ). *Function* θ *translates a formula* $(\bigvee \mathcal{X})[\Pi]$ *over* L, A, *and* P, *that is in SNF as in (1), to the (finite) initialized model* $\theta((\bigvee \mathcal{X})[\Pi]) = ((S, L, \rightarrow, A, P, \lambda_A, \lambda_P), E)$ *where* $S = \mathtt{bv}(\Pi)$, $E = \mathcal{X}$ *and for every* $X \in \mathcal{X}$ *the equation for* X *induces transitions* $\{X \xrightarrow{l} Y \mid Y \in \mathcal{Y}_{X,l}\}$, $\lambda_A(X) = B_X$, *and* $\lambda_P(X) = p$.

Theorem 5 (Maximal Model Theorem). *For any* ϕ *in SNF, we have* $\mathcal{S} \leqslant \theta(\phi)$ *if and only if* $\mathcal{S} \models \phi$.

Thus, the model $\theta(\phi)$ is a maximal model for ϕ, in the sense that $\theta(\phi)$ is a model that satisfies ϕ and simulates all models satisfying it.

5.2 Maximal Flow Graph Construction

Maximal models constructed from structural properties by the above algorithm are in general not legal flow graphs. To restrict these to legal flow graphs, we conjoin the property with a so-called *characteristic formula* C_I constructed from the interface $I = (M^+, M^-, \mathit{Modify})$. C_I describes precisely the models that constitute flow graphs with interface I:

$$\begin{aligned} C_I &= \Phi_I[\Pi_I]\,,\ \Phi_I = \textstyle\bigvee_{m\in M^+} X_m \\ \Pi_I &= \{X_m = \textstyle\bigwedge_{l\in L} [l] X_m \wedge a_m \wedge p_m \mid m \in M^+\} \\ a_m &= m \wedge \textstyle\bigwedge \{\neg m' \mid m' \in M^+, m' \neq m\} \\ p_m &= \textstyle\bigwedge \{v = v' \mid v \notin \mathit{Modify} \wedge v \in V\} \end{aligned}$$

With the help of C_I we obtain a variant of Theorem 5 for flow graphs.

Theorem 6. *Let $I = (M^+, M^-, \mathit{Modify})$ be an interface. For any initialized model $\mathcal{S} = (\mathcal{M}, E)$ over L and $A = M^+ \cup \{r\}$ we have:*

1. $\mathcal{S} \models C_I$ *if and only if* $\mathcal{R}(\mathcal{S}) : I$
2. $\mathcal{S} \leqslant_s \theta(\phi \wedge C_I)$ *if and only if* $\mathcal{S} \models \phi$ *and* $\mathcal{R}(\mathcal{S}) : I$

where $\mathcal{R}(\mathcal{S})$ denotes the reachable part of $\mathcal{S}$.

6 Compositional Verification and Tool Support

As mentioned in Section 5, for models and formulas as defined in Definition 1 and Definition 6, maximal models exist and are unique up to isomorphism. Therefore, for this choice of model and logic we can provide the following principle for compositional verification that is sound and complete for finite models: "To show $\mathcal{M}_1 \uplus \mathcal{M}_2 \models \psi$, it suffices to show $\mathcal{M}_1 \models \phi$, i.e., that component $\mathcal{M}_1$ satisfies a suitably chosen *local specification* ϕ, and $\theta(\phi) \uplus \mathcal{M}_2 \models \psi$, i.e., that $\mathcal{M}_2$, when composed with the maximal model $\theta(\phi)$, satisfies the *global property* ψ."

We exploit Theorem 6 to adapt this principle to flow graphs (as models) and structural logic and use the maximal flow graph construction from Section 5.2 to obtain the rule below.

$$\frac{\mathcal{G}_1 \models \phi \qquad \theta(\phi \wedge C_I) \uplus \mathcal{G}_2 \models \psi}{\mathcal{G}_1 \uplus \mathcal{G}_2 \models \psi} \tag{2}$$

The rule states that the composition of flow graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ satisfies the structural property ψ if flow graph $\mathcal{G}_1$ satisfies a local structural property ϕ, and the composition of flow graph $\mathcal{G}_2$ with the maximal flow graph for ϕ and interface I satisfies ψ.

Theorem 7. *Rule (2) is sound and complete.*

We restrict local specifications to structural properties, and by exploiting the fact that structural simulation implies behavioral simulation (Theorem 2), we obtain a complete compositional verification rule for global behavioral properties, thus avoiding the possibility of false negatives. However, adapting the compositional verification principle to local behavioral specifications is more problematic, as behavioral properties in general do not give rise to unique maximal flow graphs. We can represent the set of flow graphs satisfying the local specification by a (pushdown) model that simulates the behavior of these flow graphs, but this necessarily leads to approximate (i.e., sound but incomplete) solutions, since such a model cannot be guaranteed to be a legal flow graph behavior.

Tool Support and Evaluation. We have extended our compositional verification toolset [16] for the verification of PoP programs in the presence of variability. Besides the necessary data structures, the toolset includes a maximal flow graph constructor, a tool to induce behaviors from flow graphs, and external model checkers CWB [9] and Moped [18]. We used this toolset to verify a Java J2EE application consisting of 1087 lines of code, of which 297 lines are variable.

We focused on properties of *database connections*, such as "at the end of the execution, all database connections should have been closed". We therefore abstracted away all program data except variables of this type, constructed and destructed by invoking methods `getConnection` and `close`, respectively. To extract flow graphs with this abstraction, we first extracted a data-less flow graph from the Java code with our flow graph extractor tool ConFlEx [11]. Then we manually inserted all 4 database connection variables of the program into the extracted flow graphs and replaced any call to `getConnection` and `close` methods with `new` and `del` actions, respectively. This was necessary because currently we do not have a tool to extract PoP flow graphs from code. We also specified each method of the program with a structural local specification, expressing its safe sequences of invocation of methods `getConnection` and `close` (here renamed to `new` and `del`). We then (i) model checked the flow graphs of variable components against their corresponding local specifications with CWB (took 0.5 sec.), and (ii) constructed maximal flow graphs from the local specifications of the variable components (took 4.1 sec.), composed them with the flow graphs of the other components and model checked the result against a property of database connection with Moped (took 2.1 sec.). Recall that to re-verify the program after a change in the variable components only sub-task (i) needs to be repeated.

7 Related Work

In the context of compositional verification of temporal properties, the maximal model technique was first proposed by Grumberg and Long for ACTL, the universal fragment of CTL [12], and later generalized by Kupferman and Vardi for ACTL* [20]. These works do not address the verification of infinite state systems. In our previous work, we used maximal models constructed from safety μ-calculus formulas to verify infinite state context-free behaviors, where the program data is disregarded [13]. In this work we extend our previous work to a generic framework that captures program data.

For a different class of properties, Hoare logic provides a popular framework for compositional verification of programs, (see e.g. [22]) that is technically capable of verifying programs with variability. Also, of particular interest to our technique is the work by Alur and Chaudhuri [3], which proposes a unification of Hoare logic and Manna-Pnueli-style temporal reasoning by defining a set of proof rules for the verification of some particular classes of (non-regular) temporal properties. Our technique is partially inspired by this work.

Related to our approach of relativizating global properties on local specifications, Andersen introduces partial model checking in which global properties of concurrent systems are reduced to local properties of their components (processes) [5]. The work only considers finite-state systems; however, the approach suggests the possibility of extending our technique to generate local properties for variable components of programs when the global properties are fixed.

Several successful tools and techniques exist for (non-compositional) verification of behavioral properties of procedural programs. However, as mentioned, compositionality is essential for the verification of variable programs. Still, related to our two-step verification procedure, tools such as SLAM [7] and ESP [10] divide the verification into (local) intra- and (global) inter-procedural analysis to achieve scalability. It is interesting to explore if the ideas presented here can be used to adapt these tools for the verification of systems with variability.

Closely related to our flow graph model are *recursive state machines* [2], defined by Alur and others as a formalism to model procedural programs with recursive calls. The authors propose efficient LTL and CTL* model checking algorithms. However, they do not address compositional verification.

As for specification languages, the temporal logic of nested calls and returns [4] and its generalization to nested words [1] are of particular interest to this work. These logics are capable of abstracting internal computations by moving from a call to its corresponding return point in one step. However, they do not make a clear separation of structure and behavior, and may therefore require more involved maximal model constructions.

8 Conclusion

This paper presents a generic framework for compositional verification of temporal safety properties of sequential procedural programs in the presence of variability. The framework is a generalization of a previously developed framework which disregards program data. Our technique relies on local specifications of the variable components, in that the correctness of global properties of the program is relativized on the composition of the maximal flow graphs constructed from these local specifications and the flow graphs of the stable components.

The framework is parametric on a set of selected "visible" program instructions that are explicitly represented as transition labels, while the effect of all other instructions is captured abstractly by means of Hoare-style state assertions. This distinction allows to keep the level of detail of specifications within practical limits. It also allows a (symbolic) formulation of the maximal model construction for program models with data that does not add to the complexity of the construction for models without data. To evaluate our technique in practice, we provide tool support for the verification of evolving PoP programs.

In the current setting, our (symbolic) flow graphs induce behaviors with concrete data from finite domains. We conjecture that program data can be represented symbolically in the behaviors as well, using the state assertions of the structural program model (Definition 1). We plan to investigate the expressiveness of symbolic behaviors. We are currently working on a parametric flow graph extractor to extract flow graphs of Java programs for the given sets of actions and assertions. We also plan to provide tool support for the verification of programs with other datatypes, such as integers and Booleans.

References

1. R. Alur, M. Arenas, P. Barcelo, K. Etessami, N. Immerman, and L. Libkin. First-order and temporal logics for nested words. In *LICS*, pages 151–160, 2007.

2. R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, and M. Yannakakis. Analysis of recursive state machines. *TOPLAS*, 27:786–818, 2005.
3. R. Alur and S. Chaudhuri. Temporal reasoning for procedural programs. In *VMCAI*, volume 5944 of *LNCS*, pages 45–60, 2010.
4. R. Alur, K. Etessami, and P. Madhusudan. A temporal logic for nested calls and returns. In *TACAS*, volume 2998 of *LNCS*, pages 467–481. Springer, 2004.
5. H. Andersen. Partial model checking. In *LICS*, pages 398–407, 1995.
6. A. Arnold and D. Niwiński. *Rudiments of μ-calculus*, volume 146 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Publishing, 2001.
7. T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL*, pages 1–3, 2002.
8. H. Bekič. Definable operators in general algebras, and the theory of automata and flowcharts. Technical report, IBM Laboratory, 1967.
9. R. Cleaveland, J. Parrow, and B. Steffen. A semantics based verification tool for finite state systems. In *IFIP WG6.1*, pages 287–302, 1990.
10. M. Das, S. Lerner, and M. Seigle. ESP: Path–sensitive program verification in polynomial time. In *PLDI*, pages 57–68. ACM, 2002.
11. P. Gomes, A. Picoco, and D. Gurov. Sound control flow graph extraction from incomplete java bytecode programs. In *FASE*, volume 8411 of *LNCS*, pages 215–229, 2014.
12. O. Grumberg and D. Long. Model checking and modular verification. *TOPLAS*, 16(3):843–871, 1994.
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. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32:137–161, 1985.
15. M. Huisman, I. Aktug, and D. Gurov. Program models for compositional verification. In *ICFEM*, volume 5256 of *LNCS*, pages 147–166. Springer, 2008.
16. M. Huisman and D. Gurov. CVPP: A tool set for compositonal verification of control-flow safety properties. In *FoVeOOS*, volume 6528, pages 107–121, 2010.
17. Servlet Development. Release 2 (9.0.3) `http://docs.oracle.com/cd/A97688_16/generic.903/a97680/develop.htm#1007089`.
18. S. Kiefer, S. Schwoon, and D. Suwimonteerabuth. Moped–a model-checker for pushdown systems. `http://www.informatik.uni-stuttgart.de/fmi/szs/tools/moped/`.
19. D. Kozen. Results on the propositional μ-calculus. *TCS*, 27:333–354, 1983.
20. O. Kupferman and M. Vardi. An automata-theoretic approach to modular model checking. *TOPLAS*, 22(1):87–128, 2000.
21. K. Larsen. Modal specifications. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 232–246. Springer, 1989.
22. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer, 2002.
23. J. Rot, F. S. de Boer, and M. M. Bonsangue. Unbounded allocation in bounded heaps. In *FSEN*, volume 8161 of *LNCS*, pages 1–16. Springer, 2013.
24. S. Soleimanifard and D. Gurov. Algorithmic verification of procedural programs in the presence of code variability. Technical report, KTH, 2013. `http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-128950`.
25. S. Soleimanifard, D. Gurov, and M. Huisman. Procedure-modular specification and verification of temporal safety properties. *Software & Systems Modeling*, 2013. doi: 10.1007/s10270-013-0321-0.
26. C. Stirling. *Modal and Temporal Logics of Processes*. Springer, 2001.