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. To exemplify its use, we instantiate our framework to develop a compositional verification technique for programs written in a procedural language with pointers as the only datatype. We also adapt our previously developed toolset to provide support for compositional verification of such programs.

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* [18]. 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 [19,21]. 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 [33], 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.

Additionally, we present three instantiations of our generic verification framework. The first instantiation abstracts away all data as in the original CVPP framework. The second one is an instantiation for the verification of *Boolean programs* [10]. The capability of handling Boolean data shows that our generic framework can handle data from finite domains. In a third and most challenging instantiation we exemplify the use of our framework for the verification of programs written in a procedural language with pointers as the only datatype (PoP). Dealing with this language is challenging because, in addition to unbounded call stacks, it can give rise to infinite state spaces for yet another reason, namely unbounded pointer creation. This instantiation shows how our framework can cope with data from infinite domains.

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 temporal 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, (iv) an instantiation of the generic framework for a procedural language with pointer datatype (PoP), and (v) tool support for an instantiation of the framework to PoP programs.

In the remainder of this paper, Section 2 provides an overview of our technique and illustrates some variability scenarios on an example. Sections 3, 4, and 5 define our program models, specification languages, and maximal models. In Section 6 we spell-out our compositional verification principle. Sections 7, 8, and 9 present the instantiations of our framework with full data abstraction, for Boolean programs, and for PoP programs, respectively. Finally, in Section 10 we discuss related work and draw conclusions.

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 [31]. 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

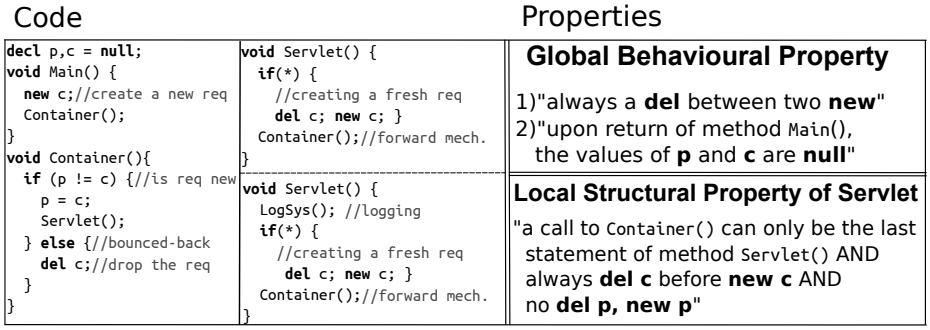


Fig. 1: Web Server Application

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. The formal instantiation of our generic verification framework to the PoP language is given in Section 9.

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 [23], 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.

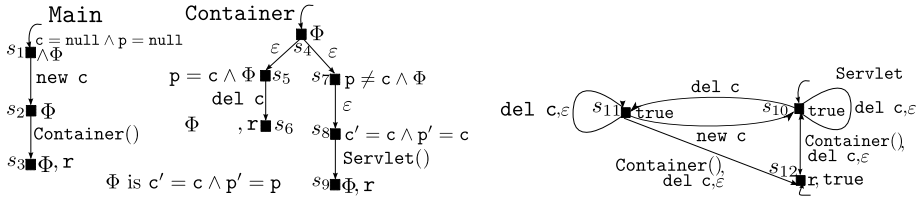
Flow Graphs. A flow graph is a finite collection of *method graphs*, each of which represents the control flow structure of a method. Our flow graphs are parametric on the class of program instructions that need to be explicitly represented for the verification of the properties of interest, while using 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 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

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



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

Fig. 2

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 verification 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., [18,26,19], 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. As we shall see in Section 9, the second global behavioral property given in Figure 1 does not hold for the local specification of method `Servlet` and flow graph of methods `Container` and `Main`.

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 models $\mathcal{M}_1$ and $\mathcal{M}_2$ is defined as simulation on their disjoint union $\mathcal{M}_1 \uplus \mathcal{M}_2$. The transitions of $\mathcal{M}_1 \uplus \mathcal{M}_2$ are defined by $in_i(s) \xrightarrow{a} in_i(s')$ if $s \xrightarrow{a} s'$ in $\mathcal{M}_i$ and its valuation by $\lambda(in_i(S)) = \lambda_i(S)$, where in_i (for $i \in \{1, 2\}$)

injects S_i into $S_1 \uplus S_2$. 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 on $\mathcal{M}_1 \uplus \mathcal{M}_2$ such that for each $s \in E_1$ there is some $t \in E_2$ with $(in_1(s), in_2(t)) \in R$. Initialized model $\mathcal{S}_1$ is simulation equivalent to $\mathcal{S}_2$, written $\mathcal{S}_1 \simeq \mathcal{S}_2$ if $\mathcal{S}_1 \leqslant \mathcal{S}_2$ and $\mathcal{S}_2 \leqslant \mathcal{S}_1$. We extend disjoint union to initialized models (by $(\mathcal{M}_1, E_1) \uplus (\mathcal{M}_2, E_2) = (\mathcal{M}_1 \uplus \mathcal{M}_2, E_1 \uplus E_2)$).

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 $\mathcal{S}_1 \uplus \mathcal{S}_2 \leqslant \mathcal{S}_1' \uplus \mathcal{S}_2'$.*

Proof. Suppose R_1 and R_2 are witnesses of $\mathcal{S}_1 \leqslant \mathcal{S}_1'$ and $\mathcal{S}_2 \leqslant \mathcal{S}_2'$, respectively. Then $R = \{((s,i),(t,i)) \mid i \in \{1,2\} \wedge (s,t) \in R_i\}$ is a simulation between $\mathcal{S}_1 \uplus \mathcal{S}_2$ and $\mathcal{S}_1' \uplus \mathcal{S}_2'$.

Next, we define formally our *flow graphs* for representing program structure. Here, for the sake of simplicity, we only consider one datatype in our formalizations. However, in a more general setting, where the program has different datatypes, a set of state assertions is defined for each datatype. The results easily generalize for this case.

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 $L_{call} = \{m(a_1, \ldots, a_w) \mid m \in \mathit{Meth}\}$ are labels representing method invocations. where a_i is an actual parameter for formal parameter p_i

Definition 3 (Method Graph). *A* method graph *for method name $m \in \mathit{Meth}$ over a set $M \subseteq \mathit{Meth}$ 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.

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. The *composition* of two interfaces is defined as their disjoint union. 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.

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,* $L_b = L_{\mathcal{A}} \cup \{m_1 \ \texttt{call} \ l_{m_2} \mid m_1 \in M^+ \wedge l_{m_2} \in L_{call}\} \cup \{m_1 \ \texttt{ret} \ m_2 \mid m_1, m_2 \in M^+\}$, $A_b = A \cup (\Sigma \times \Sigma)$, $P_b = \{\mathtt{tt}\}$, $\lambda_{A_b}(\langle s, \sigma, \sigma' \rangle, \gamma) = \lambda_A(s) \cup \{(\sigma, \sigma')\}$, *and* $\rightarrow_b \subseteq S_b \times L_b \times S_b$ *is defined by the following transition rules:*

[α] $\langle s_1, \sigma_1, \sigma_1' \rangle, \gamma) \xrightarrow{\alpha(\sigma'(a_1), \ldots, \sigma'(a_n))} (\langle s_2, \sigma_2, \sigma_2' \rangle, \gamma)$ *if* $s_1 \xrightarrow{\alpha(a_1, \ldots, a_n)} s_2 \wedge$ $(\sigma_1, \sigma_1') \models \lambda_P(s_1) \wedge$ $(\sigma_2, \sigma_2') \models \lambda_P(s_2) \wedge$ $\sigma_2 = [\![\alpha]\!]\sigma_1'$

[call] $(\langle s_1, \sigma_1, \sigma_1' \rangle, \gamma) \xrightarrow{m \ \texttt{call} \ m'(\sigma'(a_1), \ldots, \sigma'(a_w))} (\langle s, \sigma, \sigma' \rangle, \langle s_2, \sigma_1' \rangle \cdot \gamma)$ *if* $s_1 \xrightarrow{m'(a_1, \ldots, a_w)} s_2 \ \wedge \ s \models m' \wedge$ $s_1, s_2 \models m \ \wedge \ m, m' \in M^+ \wedge$ $(\sigma_1, \sigma_1') \models \lambda_P(s_1) \wedge$ $(\sigma, \sigma') \models \lambda_P(s) \wedge$ $\sigma_2 = [\![\mathit{call}]\!]\sigma_1'$

$$[ret]\ (\langle s_1, \sigma_1, \sigma_1' \rangle, \langle s_2, \sigma_2 \rangle \cdot \gamma) \xrightarrow{m\ ret(ret)\ m'} (\langle s_2, \sigma_3, \sigma_3' \rangle, \gamma)\ \textbf{if}\ s_1 \models r \wedge m \wedge \texttt{ret} = v, m' \in M^+ \wedge s_2 \models m' \wedge (\sigma_1, \sigma_1') \models \lambda_P(s_1) \wedge s_1 \models m \wedge (\sigma_3, \sigma_3') \models \lambda_P(s_2) \wedge \sigma_3 = [\![ret]\!](\sigma_1', \sigma_2)$$

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 [19].

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* $b(A) \leqslant_b b(B)$.

Proof. Let R_s be a structural simulation between $A = (\mathcal{M}^A, E^A)$ and $B = (\mathcal{M}^B, E^B)$. We lift R_s on the structural level to R_b on the behavioral level by defining:

$$\begin{aligned} R_b = \{((\langle s_1^A, \sigma_1^A, \sigma'^A \rangle, \gamma^A), (\langle s_1^B, \sigma_1^B, \sigma_1'^B \rangle, \gamma^B)) \mid \\ (s_1^A, s_1^B) \in R_s \wedge \sigma_1^A = \sigma_1^B \wedge \sigma_1'^A = \sigma_1'^B \wedge \\ (\langle s_1^A, \sigma_1^A, \sigma'^A \rangle, \gamma^A) \in b(A) \wedge (\langle s_1^B, \sigma_1^B, \sigma_1'^B \rangle, \gamma^B) \in b(B) \wedge \\ |\gamma^A| = |\gamma^B| \wedge \forall 1 \leq i \leq |\gamma^A| \,.\, \gamma_i^A = \langle s^A, \sigma^A \rangle \wedge \\ \gamma_i^B = \langle s^B, \sigma^B \rangle \wedge (s^A, s^B) \in R_s \wedge \sigma^A = \sigma^B \} \end{aligned}$$

We show that R_b is a behavioral simulation between $b(A)$ and $b(B)$.
For every entry point $s^A \in E^A$ there is an entry point $s^B \in E^B$ such that $(s^A, s^B) \in R_s$ and thus $\lambda_P(s^A) \sqsubseteq \lambda_P(s^B)$. Therefore, for each initial state $(\langle s^A, \sigma^A, \sigma'^A \rangle, \epsilon)$, there is an initial state $(\langle s^B, \sigma^B, \sigma'^B \rangle, \epsilon)$ such that

$$((\langle s^A, \sigma^A, \sigma'^A \rangle, \epsilon), (\langle s^B, \sigma^B, \sigma'^B \rangle, \epsilon)) \in R_b.$$

Suppose that $((\langle s_1^A, \sigma_1^A, \sigma_1'^A \rangle, \gamma), (\langle s_1^B, \sigma_1^B, \sigma_1'^B \rangle, \gamma)) \in R_b$. We now proceed by case analysis on the possible transitions from $(\langle s_1^A, \sigma_1^A, \sigma_1'^A \rangle, \gamma)$.

Case 1. (Actions) Suppose $(\langle s_1^A, \sigma_1^A, \sigma_1'^A \rangle, \gamma) \xrightarrow{\alpha(\sigma_1'^A(a_1), \ldots, \sigma_1'^A(a_n))} (\langle s_2^A, \sigma_2^A, \sigma_2'^A \rangle, \gamma)$. It follows that there is a state s_2^B such that $(s_2^A, s_2^B) \in R_s$, hence $\lambda_P(s_2^A) \sqsubseteq$

$\lambda_P(s_2^B)$ and $s_1^B \xrightarrow{\alpha(a_1,\ldots,a_n)} s_2^B$. Thus, for state $(\langle s_2^A, \sigma_2^A, \sigma_2^{'A}\rangle, \gamma)$ there is a state $(\langle s_2^B, \sigma_2^B, \sigma_2^{'B}\rangle, \gamma)$ such that

$$((\langle s_2^A, \sigma_2^A, \sigma_2^{'A}\rangle, \gamma), (\langle s_2^B, \sigma_2^B, \sigma_2^{'B}\rangle, \gamma)) \in R_b.$$

Case 2. (Call) Suppose

$$(\langle s_1^A, \sigma_1^A, \sigma_1^{'A}\rangle, \gamma) \xrightarrow{m' \texttt{ call } m(\sigma_1^{'A}(a_1),\ldots,\sigma_1^{'A}(a_n))} (\langle s_{1m}^A, \sigma_{1m}^A, \sigma_{1m}^{'A}\rangle, \langle s_2^A, \sigma_1^{'A}\rangle \cdot \gamma).$$

(i) We know that $s_{1_m}^A \in E_A$ and $s_{1_m}^A \models m$. Also since $s_{1_m}^A$ is an entry point, there is an entry point $s_{1_m}^B \in E_B$ such that $(s_{1_m}^A, s_{1_m}^B) \in R_s$.

(ii) We know $s_1^A \xrightarrow{m(a_1,\ldots,a_m)} s_2^A$ and that there is s_1^B such that $(s_1^A, s_1^B) \in R_s$. It follows that $s_1^B \xrightarrow{m(a_1,\ldots,a_m)} s_2^B$ such that $\lambda_P(s_1^A) \sqsubseteq \lambda_P(s_1^B)$ and $\lambda_A(s_1^A) = \lambda_A(s_1^B)$.

From (i) and (ii) we conclude that there is a transition

$$(\langle s_1^B, \sigma_1^B, \sigma_1^{'B}\rangle, \gamma) \xrightarrow{m' \texttt{ call } m(\sigma_1^{'B}(a_1),\ldots,\sigma_1^{'B}(a_n))} (\langle s_{1m}^B, \sigma_{1m}^B, \sigma_{1m}^{'B}\rangle, \langle s_2^B, \sigma_1^{'B}\rangle \cdot \gamma)$$

such that

$$((\langle s_{1m}^A, \sigma_{1m}^A, \sigma_{1m}^{'A}\rangle, \langle s_2^A, \sigma_1^{'A}\rangle \cdot \gamma), (\langle s_{1m}^B, \sigma_{1m}^B, \sigma_{1m}^{'B}\rangle, \langle s_2^B, \sigma_1^{'B}\rangle \cdot \gamma)) \in R_b$$

Case 3. (Ret) Suppose $(\langle s^A, \sigma^A, \sigma^{'A}\rangle, \gamma^A) \xrightarrow{m' \texttt{ret}(x)m} (\langle s_2^A, \sigma_2^A, \sigma_2^{'A}\rangle, \gamma^{'A})$. We know $s^A \models r \wedge m'$ and that there is s^B such that $(s^A, s^B) \in R_s$. Hence $s^B \models r \wedge m'$, $\lambda_P(s^A) \sqsubseteq \lambda_P(s^B)$, and $\lambda_A(s^A) = \lambda_A(s^B)$. We also know that $\gamma^A = (\langle s_2^A, \sigma^{''A}\rangle, \gamma^{'A})$ for some $\sigma^{''A}$. Since $((\langle s^A, \sigma^A, \sigma'^A\rangle, \gamma^A), (\langle s^B, \sigma^B, \sigma^{'B}\rangle, \gamma^B)) \in R_b$, there is a $\sigma^{''B} = \sigma^{''A}$ such that, $\gamma^B = (\langle s_2^B, \sigma^{''B}\rangle, \gamma^{'B})$, $(s_2^A, s_2^B) \in R_s$, and $\sigma^{''A} = \sigma^{''B}$. Thus, there exists a transition

$$(\langle s^B, \sigma^B, \sigma^{'B}\rangle, \gamma^B) \xrightarrow{m'' \texttt{ ret}(x)m} (\langle s_2^B, \sigma_2^B, \sigma_2^{'B}\rangle, \gamma^{'B})$$

such that

$$((\langle s_2^A, \sigma_2^A, \sigma_2^{'A}\rangle, \gamma^{'A}), (\langle s_2^B, \sigma_2^B, \sigma_2^{'B}\rangle, \gamma^{'B})) \in R_b$$

Thus we conclude that $A \leqslant_b B$. □

4 Logic

As a property specification language we use the safety fragment of Modal Equation Systems [27], that is without diamond modalities. This logic is equal in expressive power to the safety fragment of the modal μ-calculus [25]. 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 7 below is based on Bekič's principle described in [13,8].

The translation in the other direction is straightforward and done simply by replacing each fixed point by an equation.

Following Larsen [27], 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 7. *Basic simulation logic* is a variant of Hennessy-Milner logic [20] 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.

Definition 6 (Basic Simulation Logic: Semantics). *The* semantics *of a formula ϕ of basic simulation logic over L, P, and A w.r.t. model M and an environment $\rho : \mathcal{V} \to 2^S$ is defined inductively by*

$$\begin{array}{ll}
\|a\|\rho & \stackrel{def}{=} \{s \in S \mid \lambda_A(s) = a\} \\
\|\neg a\|\rho & \stackrel{def}{=} S \setminus \|a\|\rho \\
\|p\|\rho & \stackrel{def}{=} \{s \in S \mid \lambda_P(s) \sqsubseteq p\} \\
\|\neg p\|\rho & \stackrel{def}{=} S \setminus \|p\|\rho \\
\|X\|\rho & \stackrel{def}{=} \rho(X) \\
\|\phi_1 \wedge \phi_2\|\rho & \stackrel{def}{=} \|\phi_1\|\rho \cap \|\phi_2\|\rho \\
\|\phi_1 \vee \phi_2\|\rho & \stackrel{def}{=} \|\phi_1\|\rho \cup \|\phi_2\|\rho \\
\|[l]\phi\|\rho & \stackrel{def}{=} \{s \in S \mid \forall t \in S. s \xrightarrow{l} t \text{ implies } t \in \|\phi\|\rho\}
\end{array}$$

where $\sqsubseteq$ is the preorder defined on the set of state assertions.

Definition 7 (Modal Equation System). *A* modal equation system $\Pi = \{X_i = \Phi_i \mid i \in J\}$ *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 Φ_i is a formula of basic simulation logic over L, P, and A. The set of variables occurring in Π is partitioned into the set of* bound *variables, defined by* $\mathtt{bv}(\Pi) = \{X_i \mid i \in J\}$, *and the set of* free *variables* $\mathtt{fv}(\Pi)$.

The semantics of a closed modal equation system $\|[\Pi]\|\rho$ 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.

The semantics of a modal equation systems is defined in terms of its greatest solution. A solution of a modal equation system Π is a map $\eta : \mathtt{bv}(\Pi) \to 2^S$, assigning to each variable $X \in \mathtt{bv}(\Pi)$ a set of states, such that all equations in Π are satisfied. Maps η are ordered by point-wise inclusion. We first define the environment update $\rho[\eta]$, as $\rho[\eta](X) = \eta(X)$ if $X \in \mathtt{bv}(\Pi)$ and $\rho[\eta](X) = \rho(X)$ otherwise. Then we define the map $\Psi_{\Pi,\rho} \colon 2^S_{\mathtt{bv}(\Pi)} \to 2^S_{\mathtt{bv}(\Pi)}$ induced by the equations in Π by $\Psi_{\Pi,\rho}(\eta)(X) = \|\phi_X\|\rho[\eta]$.

Definition 8 (Solutions). *A* solution *of a modal equation system Π with respect to a model $\mathcal{M}$ and an environment ρ is a map $\eta : \mathtt{bv}(\Pi) \to 2^S$ such that $\Psi_{\Pi,\rho}(\eta) = \eta$. The* semantics *of a modal equation system Π with respect to $\mathcal{M}$ and ρ, denoted $\|\Pi\|\rho$, is its greatest solution.*

Note that by the well-known Knaster-Tarski fixed point theorem [34] the greatest solution of $\Psi_{\Pi,\rho}$ always exists, since $\Psi_{\Pi,\rho}$ is a monotone map on the lattice $2^S_{\mathtt{bv}(\Pi)}$ ordered by point-wise inclusion.

Definition 9 (Simulation Logic). *The formulas of* simulation logic *over L and A are defined by $\phi[\Pi]$, where ϕ is a formula of basic simulation logic and Π is a modal equation system. The set of free and bound variables are $\mathtt{fv}(\phi[\Pi]) = (\mathtt{fv}(\phi) \cup \mathtt{fv}(\Pi)) - \mathtt{bv}(\Pi)$ and $\mathtt{bv}(\phi[\Pi]) = \mathtt{bv}(\Pi)$, respectively.*

The semantics *of $\phi[\Pi]$ with respect to model $\mathcal{M}$ and environment ρ is defined by*

$$\|\phi[\Pi]\|\rho = \|\phi\|\rho[\|\Pi\|\rho].$$

We say that a state s of a model $\mathcal{M}$ satisfies $\phi[\Pi]$, written $(\mathcal{M}, s) \models \phi[\Pi]$, if $s \in \|\phi[\Pi]\|\rho$ for all ρ. For specifications $(\mathcal{M}, E)$ we define $(\mathcal{M}, E) \models \phi[\Pi]$ if $(\mathcal{M}, s) \models \phi[\Pi]$ 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 1. 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 $X[\Pi]$, where Π 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 $X = ((\mathtt{Main} \wedge \mathtt{r}) \Rightarrow (\mathtt{p} = \mathtt{null} \wedge \mathtt{c} = \mathtt{null})) \wedge \bigwedge_{l \in L_b} [l]X$.

5 Maximal Models and Flow Graphs

To construct maximal models, we generalize our previous algorithm for models without program data [19], following closely the treatment there. We therefore only sketch our construction here, and refer the reader to [19] 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 10 (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:*

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

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 [27].

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 $\mathcal{S}_1 \models \mathcal{X}(\mathcal{S}_2)$.*

Proof. (adapted from [27]; included here for completeness) Let $\mathcal{S}_1 = (\mathcal{M}_1, E_1)$ and $\mathcal{S}_2 = (\mathcal{M}_2, E_2)$.

"$\Rightarrow$" Let Ψ be the map on $2^S_{\mathsf{bv}(\Pi)}$ induced by the equations in Π (Ψ_Π before Definition 8). In order to prove that $(\mathcal{M}_1, E_1) \models (\bigvee_{s\in E_2} X_s)[\Pi_{\mathcal{M}_2}]$ it is sufficient to show that the map η defined by $\eta(X_s) = \{t \in S_1 \mid t \leqslant s\}$ is a post-fixed point of Ψ. It then follows by fixed point induction that $\eta \subseteq \|\Pi_{\mathcal{M}_2}\|$. Also, since $\mathcal{S}_1 \leqslant \mathcal{S}_2$, we have that for each $t \in E_1$ there is some $s \in E_2$ such that $t \in \eta(X_s)$. Hence $t \in \|\Pi\|(X_s)$ and therefore $t \models \chi(\mathcal{S}_2)$.

It remains to be shown that $\eta(X_s) \subseteq \Psi(\eta)(X_s)$ for all $s \in S$. Let $t \in \eta(X_s)$, hence $t \leqslant s$. We have to establish $t \in \Psi(\eta)(X_s)$, that is,

1. $t \in \|[a] \bigvee_{s\xrightarrow{a} s'} X_{s'}\|\rho[\eta]$ for all $a \in L$,
2. $t \in \|\bigwedge_{a\in\lambda_A(s)} a \;\wedge \bigwedge_{b\notin\lambda_A(s)} \neg b\|\rho[\eta]$, and
3. $t \in \|\lambda_P(s)\|\rho[\eta]$.

For (1) suppose $t \xrightarrow{a} t'$. Since $t \leqslant s$, there is a s' such that $s \xrightarrow{a} s'$ and $t' \leqslant s'$. Hence, $t' \in \eta(X_{s'})$. Points (2) and (3) follow from $t \leqslant s$ and the definition of simulation.

"$\Leftarrow$" Let $\chi(\mathcal{S}_2) = (\bigvee \mathcal{X})[\Pi]$ with $\mathcal{X} = \{X_s \mid s \in E_2\}$ and let $\eta = \|\Pi\|\rho$ for some (arbitrary) environment ρ. We show that $R = \{(s,t) \mid s \in \eta(X_t)\}$ is a simulation between $\mathcal{M}_1$ and $\mathcal{M}_2$. The result $\mathcal{S}_1 \leqslant \mathcal{S}_2$ then easily follows. Let $(s,t) \in R$, that is, $s \in \eta(X_t)$. Then s and t satisfy the same propositions, since $s \in \|\bigwedge_{a\in\lambda_A(t)} a \;\wedge \bigwedge_{b\notin\lambda_A(t)} \neg b\|\rho$, and $s \in \|\lambda_P(t)\|\rho$. Suppose now that $s \xrightarrow{a} s'$. Since $s \in \|[a] \bigvee_{t\xrightarrow{a} t'} X_{t'}\|\rho[\eta]$, we have $s' \in \eta(X_{t'})$ for some t' with $t \xrightarrow{a} t'$. This shows that R is a simulation between $\mathcal{M}_1$ and $\mathcal{M}_2$. □

Definition 11 (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.

To translate simulation logic formulas into SNF we generalize the algorithm of [19] 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 $\bigwedge_{l \in L \wedge l \notin Labels(X)} [l]\ \mathtt{tt}$, and atoms as $\bigwedge_{a \in A \wedge a \notin Atoms(X)} (a \vee \neg a)$, 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 7 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.

Example 2. The table in Figure 3 illustrates the two steps of converting a formula of simulation logic to its SNF. The formula is of form $X1[\Pi]$ where Π is given in row 1. We let $L = \{l_1, l_2\}$, $A = \{a\}$, and $P = \{p, \top, \bot\}$ where $\top$ represents the top of the lattice ordered by $\sqsubseteq$. We show how the set of equations of this formula is changed at each step. The formula, in row 2 is translated to SNF with the data-less algorithm by carrying over the state assertions; in row 2a equations are saturated by conjoining the missing atomic propositions and labels, and in row 2b new equations are introduced for disjunctions not guarded by any box. Observe that $\mathtt{tt}$ in equations $X1$ and $X2$ in row 2a gives rise to two equations $X4$ and $X5$ in row 2b. Finally in row 3, to the equations without any state assertions, $\top$ is added.

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

Proof. The correctness and termination of the algorithm for translating data-free simulation logic formulas to their SNF form are shown in [19]. We extend this algorithm by adding an additional step to the translation. However, this change does not effect the argument of the proof. □

Definition 12 (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 $\lambda_P(X) = p$.*

1)	$X1 = [l_1]X2 \wedge a \wedge p$, $X2 = [l_2]\ \mathtt{ff}$
2a)	$X1 = [l_1]X2 \wedge [l_2]\ \mathtt{tt} \wedge a \wedge p$, $X2 = [l_1]\ \mathtt{tt} \wedge [l_2]\ \mathtt{ff} \wedge (a \vee \neg a)$
2b)	$X1 = [l_1](X2 \vee X3) \wedge [l_2](X4 \vee X5) \wedge a \wedge p$, $X2 = [l_1](X4 \vee X5) \wedge [l_2]\ \mathtt{ff} \wedge a$, $X3 = [l_1](X4 \vee X5) \wedge [l_2]\ \mathtt{ff} \wedge \neg a$, $X4 = [l_1](X4 \vee X5) \wedge [l_2](X4 \vee X5) \wedge a$, $X5 = [l_1](X4 \vee X5) \wedge [l_2](X4 \vee X5) \wedge \neg a$
3)	$X1 = [l_1](X2 \vee X3) \wedge [l_2](X4 \vee X5) \wedge a \wedge p$, $X2 = [l_1](X4 \vee X5) \wedge [l_2]\ \mathtt{ff} \wedge a \wedge \top$, $X3 = [l_1](X4 \vee X5) \wedge [l_2]\ \mathtt{ff} \wedge \neg a \wedge \top$, $X4 = [l_1](X4 \vee X5) \wedge [l_2](X4 \vee X5) \wedge a \wedge \top$, $X5 = [l_1](X4 \vee X5) \wedge [l_2](X4 \vee X5) \wedge \neg a \wedge \top$

Fig. 3: Transformation of a formula to SNF

Lemma 1. *χ and θ are each others inverse up to equivalence, that is,*

1. *$\theta(\chi(\mathcal{S})) \cong \mathcal{S}$ ($\cong$ is isomorphism[2]) for finite $\mathcal{S}$, and*
2. *$\chi(\theta(\phi)) \equiv_\alpha \phi$ ($\equiv_\alpha$ is α-convertibility) for ϕ in SNF.*

Finally we are ready to relate simulation logic to simulation.

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

Proof. Follows from Theorem 3 by Lemma 1(2). □

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.

Consequences. We mention a few consequences of Theorems 3 and 5. Let $(\mathsf{S}, \leqslant)$ be the preorder of (isomorphism classes of) *finite* models over given L and A ordered by simulation and let $(\mathsf{L}, \models)$ be the preorder of formulas of simulation logic over L and A ordered by the logical consequence relation.

Corollary 1. *χ and θ are monotone.*

Simulation preserves logical properties:

Corollary 2. *For all initialized models $\mathcal{S}_1$ and $\mathcal{S}_2$ we have $\mathcal{S}_1 \leqslant \mathcal{S}_2$ and $\mathcal{S}_2 \models \phi$ imply $\mathcal{S}_1 \models \phi$.*

The pair (χ, θ) of maps forms a Galois connection between the preorders $(\mathsf{L}, \models)$ and $(\mathsf{S}, \leqslant)$.

Corollary 3. *For any finite initialized model $\mathcal{S}$ and all formulas ϕ, we have $\mathcal{S} \leqslant \theta(\phi)$ if and only if $\chi(\mathcal{S}) \models \phi$.*

[2] Here, isomorphism means a bijection of states and transitions, but labels have to be equal.

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:

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

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 $\mathcal{S}$.

Proof. (1): "⇒" Suppose $(\mathcal{M}, E) \models C_I$. We use induction on the size of M^+.

Case $M^+ = \emptyset$. In this case $C_I \equiv \mathtt{ff}$, so the only model which satisfies this property is the empty flow graph $\emptyset_M$.

Case $M^+ = \{m\}$. Here $C_I = X_m[X_m = \bigwedge_{l\in L}[l]X_m \wedge m \wedge p_m]$. Any model satisfying this property is a single method graph and all states satisfy p_m. Thus $(\mathcal{M}, E)$ is an flow graph with interface $(\{m\}, M^-, \mathit{Modify})$.

Case $M^+ = M_1 \cup M_2$ for disjoint and non-empty M_1 and M_2. Since $(\mathcal{M}, E) \models C_I$, we know that every state in the model satisfies exactly one of the atomic predicates $m \in M^+$. We can define $(\mathcal{M}_1, E_1)$ and $(\mathcal{M}_2, E_2)$ as the restrictions of $(\mathcal{M}, E)$ *w.r.t.* $I_1 = (M_1, M^-, \mathit{Modify})$ and $I_2 = (M_2, M^-, \mathit{Modify})$. Notice that $(\mathcal{M}_1, E_1) \uplus (\mathcal{M}_2, E_2) = (\mathcal{M}, E)$. We can decompose $C_I = \Phi_{(I_1)} \wedge \Phi_{(I_2)}[\Pi_{(I_1)}, \Pi_{(I_2)}]$. By induction, $(\mathcal{M}_1, E_1) : (I_1)$ and $(\mathcal{M}_2, E_2) : (I_2)$, thus by the definition of flow graph $(\mathcal{M}, E) : I$.

The restriction to the reachable part of $\mathcal{S}$ is required, because the formula C_I does not constrain the unreachable part of $\mathcal{S}$.

"⇐" By Theorem 5, it is sufficient to show $(\mathcal{M}, E) \leqslant_s \theta(C_I)$. First, we calculate $\theta(C_I)$, which gives for each method name $m \in M^+$ the method graph $(S_m, L_m, \rightarrow_m, \{m, r\}, P, \lambda_{A_m}, \lambda_{P_m})$, where

$$\begin{array}{l} S_m \ = \{X_{m,r}, X_{m,\neg r}\} \\ \rightarrow_m \ = S_m \times L \times S_m \\ \lambda_{A_m} = \{(X_{m,r}, \{m, r\}), (X_{m,\neg r}, \{m\})\} \\ \lambda_{P_m} = \{(X_{m,r}, \{v = v' \mid v \notin \mathit{Modify} \wedge v \in \ \mathtt{Glob}\}), \\ \qquad\quad (X_{m,\neg r}, \{v = v' \mid v \notin \mathit{Modify} \wedge v \in \ \mathtt{Glob}\})\} \end{array}$$

Using the relation $R = \{(s, t) \mid \lambda_A(s) = \lambda_A(t) \wedge \lambda_P(s) \sqsubseteq \lambda_P(t)\}$, it is easy to show that any model $(\mathcal{M}, E)$ with interface I is simulated by this flow graph.

(2): By (1), for flow graph $\mathcal{G} : I$, $\mathcal{G} \models \phi[\Pi]$ if and only if $\mathcal{G} \models \phi \wedge \Phi_I[\Pi, \Pi_I]$. The result then follows from Theorem 5. □

A maximal model that is constructed for the conjunction of a formula ϕ and characteristic formula for interface I, is a maximal flow graph for I and ϕ.

6 Compositional Verification

As mentioned in Section 5, for models and formulas as defined in Definition 1 and Definition 7, 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.*

Proof. "⇒" Suppose $\mathcal{G}_1 \models \phi$ and $\theta(\phi \wedge C_I) \uplus \mathcal{G}_2 \models \psi$. By Theorem 6, and the first assumption we have $\mathcal{G}_1 \leqslant \theta(\phi \wedge C_I)$. It follows that $\mathcal{G}_1 \uplus \mathcal{G}_2 \leqslant \theta(\phi \wedge C_I) \uplus \mathcal{G}_2$ by Theorems 1 and 2. Hence, $\mathcal{G}_1 \uplus \mathcal{G}_2 \models \psi$ by Corollary 2 (instantiated to the behavioral level) and the second assumption.

"⇐" Suppose $\mathcal{G}_1 \uplus \mathcal{G}_2 \models \psi$ and set $\phi = \chi(\mathcal{G}_1)$. We have to show that $\mathcal{G}_1 \models \chi(\mathcal{G}_1)$ and $\theta(\chi(\mathcal{G}_1) \wedge C_I) \uplus \mathcal{G}_2 \models \psi$. The former follows from Theorem 3 (for $\mathcal{S}_1 = \mathcal{S}_2$, instantiated to structural level). To see the latter, we start by the observation that $\chi(\mathcal{G}_1) \wedge C_I \models \chi(\mathcal{G}_1)$. By the monotonicity of θ (Corollary 1), we get $\theta(\chi(\mathcal{G}_1) \wedge C_I) \leqslant \theta(\chi(\mathcal{G}_1))$. Lemma 1 states that $\theta(\chi(\mathcal{G}_1)) \cong \mathcal{G}_1$. Hence, using the definition of structural simulation, $\theta(\chi(\mathcal{G}_1) \wedge C_I) \leqslant_s \mathcal{G}_1$. It follows by Theorems 1 and 2 that $\theta_I(\chi(\mathcal{G}_1)) \uplus \mathcal{G}_2 \leqslant_b \mathcal{G}_1 \uplus \mathcal{G}_2$. Finally, Corollary 2 and the assumption imply that $\theta(\chi(\mathcal{G}_1) \wedge C_I) \uplus \mathcal{G}_2 \models \psi$. □

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.

7 Instantiation of the Framework with Full Data Abstraction

In this section, we present an instantiation of our generic framework when program data is abstracted away completely. We show that the resulting framework is isomorphic to our previous compositional verification framework. Thus, our generic framework is a proper generalization of the old one.

Throughout this section, we use the Java program shown in Figure 4 as a running example. This program was also used as a means of illustration in our previous work [32]. Readers can compare models and properties shown in the subsequent subsections with those of [32]. In this example we consider method `even` as a variable component. A global property and the structural local specification for method `even` are given in Figure 4. In the following subsections, we instantiate our generic framework to a concrete set of state assertions, actions and their semantics. The result will be a compositional verification technique for procedural programs that abstracts away all data.

Code	Properties

public class EvenOdd {
  public boolean **odd**(int n) {
  if (n == 0) **return** false;
  else return even(n-1);
  }
	Global Behavioural Property
	"if the program execution starts in method **even**, the first call is not to method **even** itself"

public boolean even(int n) {
  if (n == 0) **return** true;
  else return odd(n-1);
  }
}
	Local Structural Property of **even**
	"method **even** can only call method **odd** and after returning, no other method can be called"

Fig. 4: A simple Java program

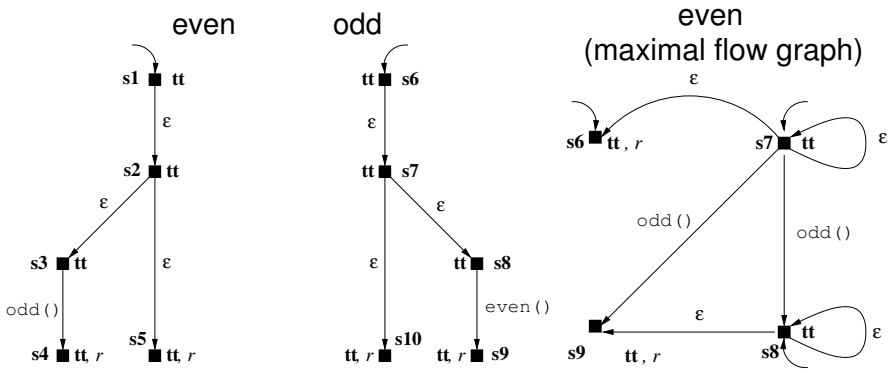
7.1 Program Models

Our flow graphs are parametrized on a set of state assertions and actions. The definition of flow graphs with full data abstraction is an instantiation of the generic definition of flow graphs with the assertions and actions defined below.

State Assertions. When the data is fully abstracted away, the set of assertions P should be empty. However, to be faithful to Definition 1, we use the (dummy) assertion `tt` and assign it to all states of data-less flow graphs. Then, the semantic entailment $\sqsubseteq$ on state assertions is defined by *equality*.

Actions. Since we are interested in control-flow properties, we do not identify any particular program instructions as actions. However, we include the neutral action ε for simplicity, and thus the set of actions is the singleton set $\{\varepsilon\}$.

Example 3. Figure 5a shows the flow graph of the program in Figure 4. Its interface is $M^+ = \{\texttt{even}, \texttt{odd}\}, M^- = \{\texttt{even}, \texttt{odd}\}, \mathit{Modify} = \{\}$. It consists of two method graphs, for methods `even` and `odd`. The interface of method `even` is $M^+ = \{\texttt{even}\}, M^- = \{\texttt{odd}\}, \mathit{Modify} = \{\}$.



(a) Flow graph of EvenOdd

(b) Maximal flow graph for method `even`

In order to instantiate the flow graph behavior we first need to define program states.

Program States. For the instantiation of our framework with full program abstraction, the sets of variables V and their values $\mathcal{D}$ are empty. However, to be faithful to our definition of program state we assume a dummy variable (v) and a single value (true) that form a single program state σ mapping v to true.

Flow Graph Behavior. The flow graph behavior with full data abstraction is simply defined as an instantiation of the generic definition of behavior (Definition 4) where the semantics of labels are *identity* function, i.e.,

$$[\![\varepsilon]\!]\sigma_1' = \sigma_1'$$

$$[\![call]\!]\sigma_1' = \sigma_1'$$

$$[\![ret]\!](\sigma_1', \sigma_2) = \sigma_2$$

Example 4. Consider the flow graph in Figure 5a. One example run through its (branching, infinite-state) behavior, for $\sigma = v \to true$, from an initial to a final configuration, is:

$$(\langle s_1, \sigma, \sigma\rangle, \epsilon) \xrightarrow{\varepsilon} (\langle s_2, \sigma, \sigma\rangle, \epsilon) \xrightarrow{\varepsilon} (\langle s_3, \sigma, \sigma\rangle, \epsilon) \xrightarrow{\texttt{even call odd()}} (\langle s_6, \sigma, \sigma\rangle, \langle s_4, \sigma, \sigma\rangle) \xrightarrow{\varepsilon}$$
$$(\langle s_7, \sigma, \sigma\rangle, \langle s_4, \sigma, \sigma\rangle) \xrightarrow{\varepsilon} (\langle s_{10}, \sigma, \sigma\rangle, \langle s_4, \sigma, \sigma\rangle) \xrightarrow{\texttt{odd ret() even}} (\langle s_4, \sigma, \sigma\rangle, \epsilon)$$

7.2 Logics

The structural and behavioral logics are also defined as instantiations of generic simulation logic (Definition 9) for the sets of assertions and actions defined above.

Example 5. Consider the local structural specification of method `even` mentioned informally in Figure 5a. This specification is formalized by the following structural formula $(\texttt{even} \Rightarrow X)[\Pi]$, where Π is:

$$X = [\texttt{even()}]\texttt{ff} \wedge [\varepsilon]X \wedge [\texttt{odd()}]Y$$
$$Y = [\texttt{even()}]\texttt{ff} \wedge [\texttt{odd()}]\texttt{ff} \wedge [\varepsilon]Y$$

The global behavioral property shown in the figure is formalized by the following behavioral formula $(\texttt{even} \Rightarrow X)[\Pi]$ where Π is:

$$X = [\texttt{even call even()}]\texttt{ff} \wedge [\varepsilon]X$$

7.3 Maximal Flow Graphs

Maximal flow graphs are constructed by applying the algorithm described in Section 5 for the set of structural labels (method names and actions), atomic propositions (method names and `r`), and the top element of the lattice of state assertions ordered by $\sqsubseteq$ (which is here `tt`).

Example 6. To construct maximal flow graph for method `even`, the following characteristic formula is constructed from its interface.

$(X0 \vee X1)[\Pi]$ where Π is:

$$X0 = ([\varepsilon](X0 \vee X1) \wedge [\texttt{even()}](X0 \vee X1) \wedge [\texttt{odd()}](X0 \vee X1) \wedge \texttt{r} \wedge \texttt{even} \wedge \texttt{tt})$$
$$X1 = ([\varepsilon](X0 \vee X1) \wedge [\texttt{even()}](X0 \vee X1) \wedge [\texttt{odd()}](X0 \vee X1) \wedge \neg\texttt{r} \wedge \texttt{even} \wedge \texttt{tt})$$

The above formula is conjoined with the local structural specification of method `even` presented above and the maximal flow graph shown in Figure 5b is constructed.

7.4 Compositional Verification

We use Rule 2 for the program models, logics and maximal flow graphs defined in sections 7.1, 7.2, and 7.3. As a result, we obtain a compositional verification technique for procedural programs where data is abstracted away.

Example 7. We use Rule 2 to verify the program in Figure 4, considering method `even` as a variable component. This divides the verification effort into the following two independent sub-tasks. (i) We check that the structural local specification of method `even` is satisfied by its implementation. (The implementation of `even` in the figure satisfies its local structural specification.) (ii) A maximal flow graph of method `even` is constructed and composed with the flow graph of method `odd`. Then the behavior induced from the resulting flow graph is model checked against the global properties of the program.

8 Instantiation of the Framework for Boolean Programs

In this section, we present an instantiation of our generic framework for Boolean programs [10]. These are procedural programs with Boolean variables as the only datatype. This language has been studied and used as abstract representation of real-life programming languages in several works [12,9,11,29,15]. Here, we first define formally the language and then instantiate the definitions of flow graph, behavior, logic, and the maximal flow graph construction algorithm to Boolean programs.

8.1 Syntax

The syntax of Boolean programs is shown in Table 1. The language has Boolean variables as the only datatype. It supports conditional statements, loops, and method calls. A method has a return value if an expression or a variable name is provided to the return statement, otherwise the method returns `false` by reaching a return statement or the end of its execution.

As shown in Table 1, the variables of Boolean programs are either global (if they are declared outside the scope of a method), local to a method (if they are declared inside the scope of a method), or parameters.

Throughout this section, we use the Boolean program shown in Figure 5 as a running example. The program resembles an electronic referendum voting system that gets a vote from a voter and registers it into an electronic ballot box, such as a database. The program has two global variables `a` and `v` both of them initialized to `false`. Variable `v` is used to store the vote and variable `a` is to store the result of the authentication performed in method `Authen`. The program starts in method `Main`. Method `GetVote` resembles a method to get a vote from a voter for example by using an input device. Method `Authen` resembles an external algorithm used for authenticating the voters. Here, however, these models randomly assign `true` or `false` to variables `v` and `a`. We assume that there are various algorithms for authentication and thus the program is delivered

Syntax			**Description**
prog	::=	*decl* proc**	A program is a list of global variable declarations followed by a list of method definitions
decl	::=	**decl** *id* **;**	Declaration of variables
	|	**decl** *id* = *val* **;**	
id	::=	[a-zA-Z_][a-zA-Z0-9_]*	An identifier is a C-style identifier
val	::=	**ture**	
	|	**false**	
proc	::=	*id* (*idlst*) { *decl* sseq* }	Function definition
idlst	::=	*id*	
	|	*id* **,** *idlst*	
sseq	::=	$stmt^+$	Sequence of statements
stmt	::=	*id* = *id* **;**	Assignment statement
	|	*id* = *val* **;**	Conditional statement
	|	**if** (*bexpr*) { *sseq* } **else** { *sseq* }	Iteration statement
	|	**while** (*bexpr*) { *sseq* }	Method call
	|	*id* (*idlst*) **;**	Return statement with return value
	|	**return** *id* **;**	Return statement
	|	**return ;**	
bexpr	::=	*bexpr binop bexpr*	
	|	**!** *bexpr*	Negation
	|	(*bexpr*)	
	|	*expr*	
binop	::=	**&&**	And
	|	||	Or
expr	::=	*id eqop id*	
	|	*id eqop val*	
	|	*****	Non-deterministic choice
eqop	::=	**==**	Equality
	|	**!=**	Inequality

Table 1: Syntax of Boolean Programs

without any specific implementation for method `Authen`; it becomes complete by adding a concrete implementation of this method depending on the preferred authentication algorithm. Two implementations for this method are given in the left column of Figure 5 in which the one at the top is a legitimate implementation and the second one is an implementation by an intruder to manipulate the voting results.

Our task is to prove that a behavioral global property holds for the program, given the structural local specifications of its variable components. In the example above, method `Authen` is a variable component. A behavioral global property and the local specification of `Authen` are given in Figure 5.

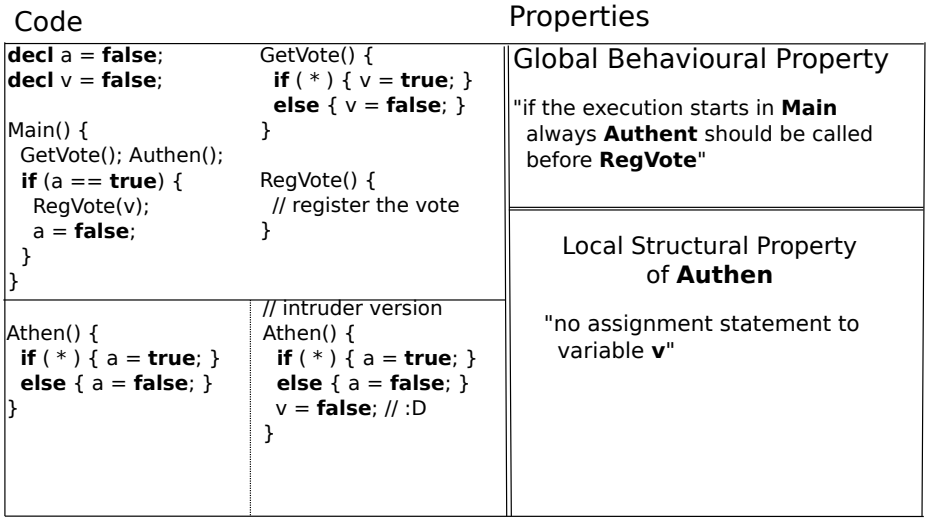


Fig. 5: Referendum voting program

8.2 Program Models

The definition of flow graphs with Boolean data is an instantiation of the definition of generic flow graphs with the assertions and actions defined below.

State Assertions. To formally define the set of state assertions of Boolean programs, we assume, w.l.o.g., that all methods have n local variables and w parameters. We represent the variables of a program by the set $V_{bool} = \texttt{Glob} \cup \texttt{Loc} \cup \texttt{Par}$, where $\texttt{Glob} = \{\texttt{g}_1, \ldots, \texttt{g}_\texttt{k}\}$ is the set of *global variables*, $\texttt{Loc} = \{\texttt{l}_1, \ldots, \texttt{l}_\texttt{n}\}$ is the set of *local variables*, and $\texttt{Par} = \{\texttt{p}_1, \ldots, \texttt{p}_\texttt{w}\}$ is the set of *parameters*. We let $v_1, \ldots, v_z$ represent the variables in V_{bool}. We analogously define the set $V'_{bool} = \texttt{Glob}' \cup \texttt{Loc}' \cup \texttt{Par}'$ as the set of *primed variables* where $\texttt{Glob}'$, $\texttt{Loc}'$, and $\texttt{Par}'$ are the sets of primed global variables, local variables, and parameters, respectively.

In addition to the variables above, we define a reserved global variable `ret` that will be used to store the return value from a method call. The variable `ret` does not have a primed version because it is not used in the program's code and we assume its value can only be changed through a return statement.

Atomic formulas *Cond* that form the assertions are equality and inequality checks over program and primed variables. They are of the form $v_1 = v_2$, $v_1 \neq v_2$, or $v_2 = val$, where $v_1 \in V_{bool}$, $v_2 \in V_{bool} \cup V'_{bool}$, and $val = \{\texttt{true}, \texttt{false}\}$. The set of assertions is $P_{bool} = \{\bigwedge C \mid C \subseteq Cond\}$. Then the preorder $\sqsubseteq$ on assertions is defined by *logical implication*.

Actions. Since we are interested in properties expressing the value of variables exactly before and after method calls, we do not identify any particular program

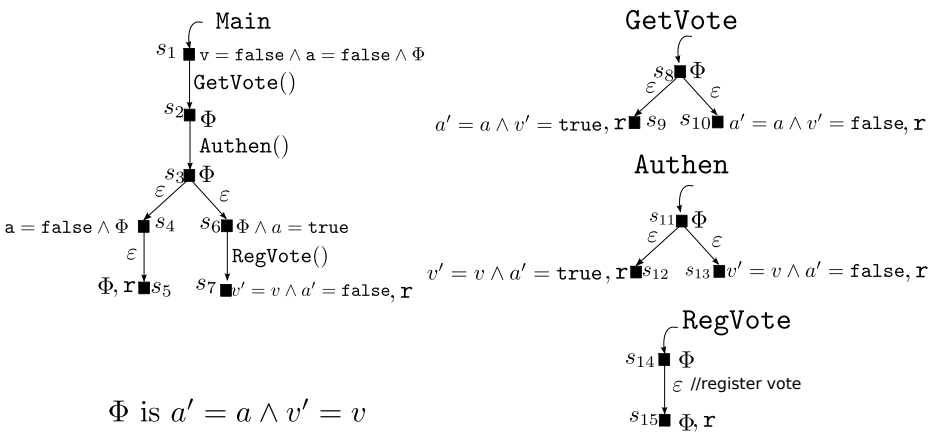


Fig. 6: Flow Graph Example for Boolean Program in Figure 5

instructions as actions. However, we include the neutral action ε for simplicity, and thus the set of actions of Boolean programs is the singleton set $\mathcal{A}_{bool} = \{\varepsilon\}$.

Example 8. Figure 6 shows a flow graph of the Boolean program shown in Figure 5. The guard of the if statement in the code of method `Main` is captured by the state assertions assigned to states s_4 and s_6, and assignment `a:=false` is captured by the assertion of state s_7. Variable `ret` is not shown in this example to simplify the presentation. Ignoring this variable does not affect the example, because methods do not return values.
The interface of method `Authen` is $M^+ = \{\texttt{Authen}\}, M^- = \{\}, \mathit{Modify} = \{a\}$.

Program States. Program states of Boolean programs are mappings from the set of variables V_{bool} to their values in $\{\texttt{false}, \texttt{true}\}$.

Flow Graph Behavior. The flow graph behavior for Boolean programs is an instantiation of Definition 4 for the set of actions $\mathcal{A}_{bool}$ and their semantics. We formalize the state updates by multiple mappings of the form $\sigma[v_i \mapsto \sigma(g_i)]_{1 \le i \le k}$ (or $\sigma[v_i \mapsto c]_{1 \le i \le k}$), which for $i \in \{1, \cdots, k\}$ updates the equivalence class of variable v_i in σ to the one of g_i in σ (or concrete value c).

$$[\![\varepsilon]\!]\sigma_1' = \sigma_1'$$

$$[\![\mathit{call}]\!]\sigma_1' = \sigma_1'[p_i \mapsto \sigma_1'(a_i)]_{1 \le i \le w} \; [l_i \mapsto \texttt{false}]_{1 \le i \le n}$$

$$[\![\mathit{ret}]\!](\sigma_1', \sigma_2) = \sigma_2[g_i \mapsto \sigma_1'(g_i)]_{1 \le i \le k} \; [\texttt{ret} \mapsto \sigma_1'(v)]$$

Example 9. An example sequential run through the branching behavior of the flow graph of Figure 6 is shown below. In the run, the program states are represented by boxes. The value of the variables in the left box are `false`, and the ones in the right box are `true`. For example $\boxed{\texttt{a}|\texttt{v}}$ shows that the value of variable `a` is `false` and the value of variable `v` is `true`.

Notice that the assertion of state s_1 (i.e., $\texttt{a} = \texttt{false} \wedge \texttt{v} = \texttt{false}$) restricts the initial program state to $\boxed{\texttt{a, v}|\,}$.

$$(\langle s_1, \boxed{\texttt{a, v}|\,}, \boxed{\texttt{a, v}|\,}\rangle, \epsilon)$$

$$\xrightarrow{\texttt{Main call GetVote()}} (\langle s_8, \boxed{\texttt{a, v}|\,}, \boxed{\texttt{a, v}|\,}\rangle, \langle s_2, \boxed{\texttt{a, v}|\,}\rangle)$$

$$\xrightarrow{\varepsilon} (\langle s_9, \boxed{\texttt{a, v}|\,}, \boxed{\texttt{a}|\texttt{v}}\rangle, \langle s_2, \boxed{\texttt{a, v}|\,}\rangle)$$

$$\xrightarrow{\texttt{GetVote ret Main}} (\langle s_2, \boxed{\texttt{a}|\texttt{v}}, \boxed{\texttt{a}|\texttt{v}}\rangle, \epsilon)$$

$$\xrightarrow{\texttt{Main call Authen()}} (\langle s_{11}, \boxed{\texttt{a}|\texttt{v}}, \boxed{\texttt{a}|\texttt{v}}\rangle, \langle s_3, \boxed{\texttt{a}|\texttt{v}}\rangle)$$

$$\xrightarrow{\varepsilon} (\langle s_{12}, \boxed{\texttt{a}|\texttt{v}}, \boxed{|\texttt{a, v}}\rangle, \langle s_3, \boxed{\texttt{a}|\texttt{v}}\rangle)$$

$$\xrightarrow{\texttt{Athen ret Main}} (\langle s_3, \boxed{|\texttt{a, v}}, \boxed{|\texttt{a, v}}\rangle, \epsilon)$$

$$\xrightarrow{\varepsilon} (\langle s_6, \boxed{|\texttt{a, v}|}, \boxed{|\texttt{a, v}|}\rangle, \epsilon)$$

$$\xrightarrow{\texttt{Main call RegVote()}} (\langle s_{14}, \boxed{|\texttt{a, v}|}, \boxed{|\texttt{a, v}|}\rangle, \langle s_7, \boxed{|\texttt{a, v}|}\rangle)$$

$$\xrightarrow{\varepsilon \text{ //register the vote}} (\langle s_{15}, \boxed{|\texttt{a, v}|}, \boxed{|\texttt{a, v}|}\rangle, \langle s_7, \boxed{|\texttt{a, v}|}\rangle)$$

$$\xrightarrow{\texttt{RegVote ret Main}} (\langle s_7, \boxed{|\texttt{a, v}|}, \boxed{\texttt{a}|\texttt{v}}\rangle, \epsilon)$$

8.3 Logics

The structural and behavioral logics are also defined as instantiations of generic simulation logic (Definition 9) for the sets of assertions and actions defined above.

Example 10. The local structural property of method `Authen` shown in Figure 5 is specified by the structural formula $(X)[\Pi]$, where Π is:

$$X = \bigwedge_{l \in L_s} [l]X \wedge \texttt{v}' = \texttt{v}$$

and $L_s = \{\varepsilon, \texttt{Authen}()\}$.

The global property in Figure 5 can be formalized in behavioral simulation logic $(\texttt{Main} \Rightarrow X)[\Pi]$, where Π is:

$$X = [\texttt{Main call RegVote}()]\texttt{ff} \wedge \bigwedge_{l \in L} [l]X$$

and $L = L_b \backslash \{\texttt{Main call RegVote}(), \texttt{Main call Authen}()\}$.

8.4 Maximal Flow Graphs

Maximal flow graphs are constructed by applying the algorithm described in Section 5 for the set of structural labels (method names and actions), atomic propositions (method names and `r`), and the top element of the lattice of state assertions ordered by $\sqsubseteq$.

Example 11. As a first step of constructing a maximal flow graph for method `Authen`, the following characteristic formula is constructed form its interface.

$(X0 \vee X1)[\Pi]$ where Π is

$$X0 = ([\varepsilon](X0 \vee X1) \wedge [\texttt{Authen()}](X0 \vee X1) \wedge \texttt{r} \wedge \texttt{Authen} \wedge v' = v)$$
$$X1 = ([\varepsilon](X0 \vee X1) \wedge [\texttt{Authen()}](X0 \vee X1) \wedge \neg\texttt{r} \wedge \texttt{Authen} \wedge v' = v)$$

The above formula is conjoined with the local structural specification of method `Authen` presented in Section 8.3 and the maximal flow graph shown in Figure 7 is constructed.

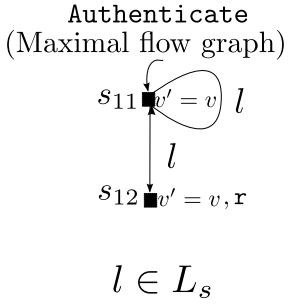


Fig. 7: Maximal Boolean flow graph of method `Authen`
$L_s = \{\varepsilon, \texttt{Authen}()\}$

Note: The characteristic formula constructed from the interface of method `Authen` only allows modifications to variable `a`, forbidding any change to variable `v`. The local property of this method essentially expresses the same constraint on the implementation of the method. For the maximal flow graph construction, where the constraints specified by interfaces and local properties are conjoined, it is sufficient to express such constraints either in interfaces or local properties.

8.5 Compositional Verification

We use Rule 2 for the program models, logics and maximal flow graphs defined in sections 8.2, 8.3, and 8.4. As a result, we obtain a compositional verification technique for these programs.

Example 12. Consider the program in Figure 5 and its properties. We use Rule 2 to verify this program, assuming method `Authen` as the only variable component. This divides the verification into the following two independent sub-tasks. (i) We check that the local specification of method `Authen` is satisfied by its implementation (whenever it is available or changed). (The implementation of `Authen` at the top satisfies its local structural specification, while the one at the bottom does not.) (ii) A maximal flow graph is constructed from the interface and local specification of method `Authen` and composed with the flow graph of the remaining methods of the program. Then the behavior induced from the resulting flow graph is model checked against the global property of the program.

9 Instantiation of the Framework for the PoP Language

In this section, we present an instantiation of our generic verification framework for Pointer Programs (PoP), that are programs with pointers as the only datatype. We instantiate the definitions of flow graph, flow graph behavior, logic, and maximal flow graph construction algorithm to the PoP language.

9.1 syntax

The syntax of the PoP language is an adaptation of the syntax of Boolean programs shown in Table 1 with two additional statements:

new *id* **;**	creating a fresh pointer and assigning it to a variable
del *id* **;**	deleting the pointer a variable pointing to

Also, in the PoP language, the concrete values for variables do not exist, i.e., variables cannot be assigned to values, and conditional statements are only equality and inequality checks on variables and non-deterministic choice. Thus, declaration **decl** *id* = *val* **;**, statement *id* = *val*, and Boolean expression *id eqop val* do not exist in the syntax of PoP language.

PoP method calls are call-by-reference. A method has a return value if an expression or a variable name is provided to the return statement, otherwise the method returns `null` by reaching a return statement or the end of its execution.

We illustrate our approach on a web application that has local variables and parameters. The program in Figure 8 is an implementation of a container and two implementations of a servlet, in which the one in the right column extends the left one by adding a logging facility through calling method `LogSys`. In the code, the variables are pointers to requests. The global variable `P` points to the last-received request, while `c` to the current one. At the beginning of the execution, the request `c` is received by `Container` and passed to `Servlet` that non-deterministically calls back `Container` with the current or a fresh request which variable `n` points to. By this, we mimic the call-backs to `Container` 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 in

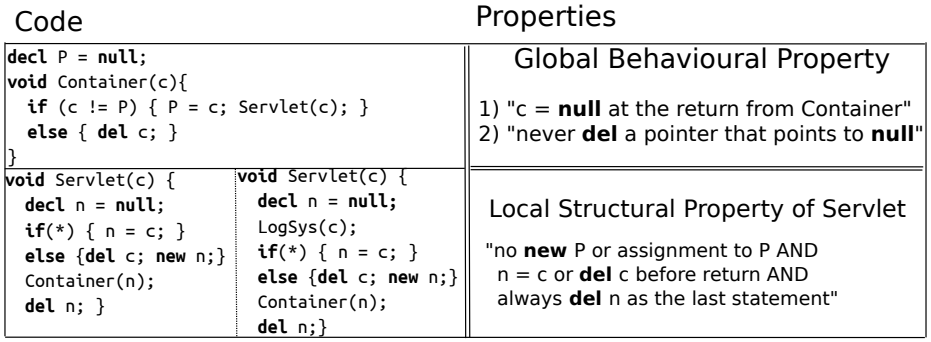


Fig. 8: Web Server Application

the figure but we assume that it does not modify the global variables. In this example, again we consider each method as a component and we consider method `Servlet` as the variable part of the program.

The structural local specifications of methods `Servlet` and `LogSys`, and a behavioral global property are given in the figure. As mentioned, in our framework, structural local specifications are more meaningful than behavioral ones. The reason is that they express properties of the syntax of programs rather than their execution, thus can independently be checked against the component's code rather than the execution of the whole program. For example, a behavioral local specification of method `Servlet` is "the parameter `c` points to `null` at any return point of method `Servlet`"[3], which cannot be checked for method `Servlet` in isolation from `Container` and `LogSys`, because the value of `c` is not clear after the return from the call to `Container`.

As we shall see, to verify the behavioral global property, the structural local specifications of methods `Servlet` and `LogSys` are checked independently (sub-task (i)), and that the composition of these local specifications with the implementation of `Container` entails the global property (sub-task (ii)).

9.2 Program Models

The definition of flow graphs for PoP programs is defined as an instantiation of our generic flow graph notion with the state assertions and actions defined below.

State Assertions. The state assertions of PoP programs are defined similarly to those of Boolean programs. Again, to formally define the set of state assertions of PoP programs, we assume, w.l.o.g., that all methods have n local variables and w parameters. Program variables are represented by the set $V_{pop} = \texttt{Glob} \cup \texttt{Loc} \cup \texttt{Par}$, where $\texttt{Glob} = \{\texttt{g}_1, \ldots, \texttt{g}_\texttt{k}\}$ is the set of *global variables*, $\texttt{Loc} = \{\texttt{l}_1, \ldots, \texttt{l}_\texttt{n}\}$ is the set of *local variables*, and $\texttt{Par} = \{\texttt{p}_1, \ldots, \texttt{p}_\texttt{w}\}$ is the set of *parameters*. We let

[3] Which gives rise to infinitely many implementations.

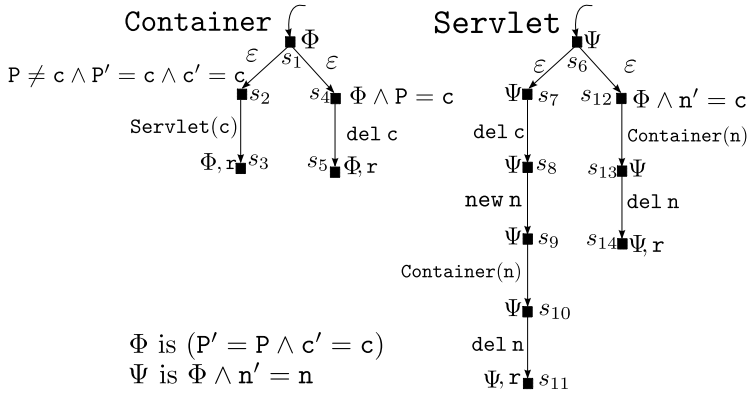


Fig. 9: Flow Graph Example

$v_1, \ldots, v_z$ represent the variables in V_{pop}. Again, we analogously define the set $V'_{pop} = \texttt{Glob}' \cup \texttt{Loc}' \cup \texttt{Par}'$ as the set of *primed variables* where $\texttt{Glob}'$, $\texttt{Loc}'$, and $\texttt{Par}'$ are the sets of primed global variables, local variables, and parameters, respectively.

In addition to the variables above, we define a reserved global variable `ret` that will be used to store the return value from a method call. The variable `ret` does not have a primed version because it is not used in the program's code and we assume that its value can only be changed by a return statement.

Atomic formulas *Cond* that form the logical assertions are equality and inequality checks over program and primed variables of the form $v_1 = v_2$ or $v_1 \neq v_2$, where $v_1 \in V_{pop}$ and $v_2 \in V_{pop} \cup V'_{pop}$. The set of assertions is $P_{pop} = \{\bigwedge C \mid C \subseteq \mathit{Cond}\}$. Then the preorder $\sqsubseteq$ on assertions is defined by logical entailment.

Actions. The actions of PoP programs are pointer creation and deletion, and a neutral action denoted by ε, thus $\mathcal{A}_{pop} = \{\texttt{del}, \texttt{new}, \varepsilon\}$.The arity of `del` and `new` is one and of ε is zero.

Example 13. Figure 9 shows a flow graph of the PoP program shown in Figure 8. The guard of the if statement and assignment `P:=c` in the code of method `Container` are captured by the state assertions assigned to states s_2 and s_4. Again, variable `ret` is not shown in this example to simplify the presentation. Ignoring this variable does not affect the example, because methods do not return values.
The interface of method `Servlet` is $M^+ = \{\texttt{Servlet}\}, M^- = \{\texttt{Container}, \texttt{LogSys}\}, \mathit{Modify} = \{\}$, where *Modify* is empty, because in the code of method `Servlet` there is no syntactic assignment to the global variable `P`.

Program State. PoP programs can create infinite state spaces for two reasons: unbounded call stacks, and infinite memory allocation. To be able to model check

the behavioral models of such programs, we address the former by modeling the unbounded call stacks as pushdown automata/systems and exploit existing model checking algorithm such as [24]. For the latter however, we abstractly represent the infinite memory allocations by finitely many program states. To achieve this, we exploit the fact that at any point of execution (behavior) of PoP programs, only finitely many memory locations are referenced by program variables. This allows us to represent the allocated memories as a partitioning of variables into equivalence classes. The idea is that two variables are in one equivalence class if they are pointing to the same memory. Then the number of equivalence classes is at most equivalent to the number of variables of the program (when all variables are distinct). Such an abstraction is adequate to capture the behavior of PoP programs because only equality and inequality checks of variables are allowed by the language.

To implement this abstraction, we use an extension of the technique proposed by Rot et al. in [30]. They use a set of so-called "freeze" variables $\mathcal{F}_g = \{g_1^f, \ldots, g_k^f\}$ that consists of one additional local variable for each global variable of the program. These variables are used to store the values of global variables at each method invocation and as a means of recomputation of their values after returns from method calls (see the definition of function $\mathfrak{G}$ in the flow graph behavior). The global variable `ret` does not have a freeze variable. We introduce an additional set of freeze variables for parameters $\mathcal{F}_p = \{p_1^f, \ldots, p_w^f\}$ to provide support for call-by-reference parameter passing. These variables store the values of parameters at the method calls and are used to compute the values of local variables at the returns (see the definition of function $\mathfrak{L}$ in the flow graph behavior).

In addition to variables $\mathcal{F}_g$ and $\mathcal{F}_p$, program states include a set of so called "indicator" variables $\mathbb{D} = \{d_1, \ldots, d_{k+w}\}$ that consists of one Boolean variable for each global variable or parameter. These are used to provide support for delete instructions. Let us explain by an example why these variables are needed. Consider the following code.

```
1 decl x;
2 Caller() {
3  decl l;
4  new x;
5  l = x;
6  Callee();
7 }
```

```
8 Callee() {
9  decl l2;
10 l2 = x;
11 del l2;
12 }
```

In order to have an accurate partitioning of variables, we need to be able to find out the value of local variable `l` upon returning from a call to method `Callee` (line 7). Assume that `l` and `x` point to memory m before the call. Since `l` is local to `Caller`, its value cannot be directly modified in `Callee`. Still, in `Callee`, m can be deleted through deleting the global variable `x`. We set variables in $\mathbb{D}$ to `true` if a deleted variable in the current context points to a memory that is referenced in the previous context. This is checked at every delete operation and the result is stored as a Boolean value in variables in $\mathbb{D}$ (see the definition

of $[\![del]\!]$ in the flow graph behavior). For example, to construct the behavioral model for the code above, while line 11 is executing, the variable in $\mathbb{D}$ that corresponds to global variable `x` is set to `true`. Upon a return from a call, the values of these variables are used to find out if the memory a local variable is pointing to has been deleted during the execution of the called method, through a global variable or parameter (see the definition of function $\mathfrak{L}$ in the flow graph behavior).

Thus, a program state of PoP programs consists of two parts:

1. A partitioning over the set of program and freeze variables $V^f_{pop} = V_{pop} \cup \mathcal{F}_g \cup \mathcal{F}_p$. We define such partitionings by assigning to each variable a natural number that represents its equivalence class.
2. A mapping of variables in $\mathbb{D}$ to `true` or `false`.

As a result, program states are defined formally as $\Sigma : V^f_{pop} \to \{n \mid 0 \leq n \leq |V^f_{pop}| + 1\} \times \mathbb{D} \to \{\texttt{true}, \texttt{false}\}$. By this formulation, two distinct variables v_1 and v_2 belong to the same equivalence class iff $\sigma(v_1) = \sigma(v_2)$. We use 0 for the equivalence class of `null` (also denoted by $\perp$).

Flow Graph Behavior. The flow graph behavior for PoP programs is an instantiation of Definition 4 for the set of actions $\mathcal{A}_{pop}$ and their semantics. To define formally the semantics of actions, we formalize state updates by multiple mappings of the form $\sigma[v_i \mapsto \sigma(g_i)]_{1 \leq i \leq k}$ (or $\sigma[v_i \mapsto c]_{1 \leq i \leq k}$), which updates for $i \in \{1, \cdots, k\}$ the equivalence class of variable v_i in σ to the one of g_i (or the concrete class c). We sometimes use conditional updates of the form $\sigma[b \,?\, update_1 : update_2]$ that is $\sigma[update_1]$ if b is evaluated to *true* and $\sigma[update_2]$ otherwise.

Intuitively, the semantics of ε is the identity function, `del(`v`)` moves v and all of its aliases to the equivalence class of `null` (i.e., 0) and sets the values of variables in $\mathbb{D}$, `new(`v`)` maps v to a fresh equivalence class (through using function $\mathfrak{N}$), *call* initializes a program state for the called method, and *ret* is defined through the functions $\mathfrak{L}$ and $\mathfrak{G}$.

Function $\mathfrak{N}(v, \sigma)$ returns the least natural number between 1 and $|V^f_{pop}| + 1$ which is not used in program state σ, if all numbers are used it returns $\sigma(v)$. Other functions ($\mathfrak{R}$, $\mathfrak{L}$, $\mathfrak{N}$, and $\mathfrak{G}$) are formally defined below. They all return a natural number. Function $\mathfrak{R}$ computes the value of variable `ret` upon a return from a call. Function $\mathfrak{L}$ copies the values of the local variables from the state before the call to the state after its return. The value of a local variable is 0 if it is an alias for a global variable that is deleted in the called method (see Example 14). Function $\mathfrak{G}$ recursively computes the equivalence classes of global variables on a return from a method call based on their equivalence classes before the call and at the return of the call, as in [30]. In short, it moves a global variable to either the equivalence class of `null`, or of another variable, or to a fresh equivalence class, if it is deleted, assigned to another variable in the called method, or assigned to a new pointer, respectively.

$$
\begin{aligned}
[\![\varepsilon]\!]\sigma_1' &= \sigma_1' \\
[\![\texttt{new}(v)]\!]\sigma_1' &= \sigma_1'[v \mapsto \mathfrak{N}(v, \sigma_1')] \\
[\![\texttt{del}(v)]\!]\sigma_1' &= \sigma_1'[\sigma_1'(v) = \sigma_1'(l_i)\ ?\ l_i \mapsto 0 : li \mapsto \sigma_1'(l_i)]_{1\le i\le n} \\
&\quad\ [\sigma_1'(v) = \sigma_1'(g_i)\ ?\ g_i \mapsto 0 : g_i \mapsto \sigma_1'(g_i)]_{1\le i\le k} \\
&\quad\ [\sigma_1'(v) = \sigma_1'(p_i)\ ?\ p_i \mapsto 0 : p_i \mapsto \sigma_1'(p_i)]_{1\le i\le w} \\
&\quad\ [\sigma_1'(v) = \sigma_1'(g_i^f)\ ?\ d_i \mapsto \texttt{tt} : d_i \mapsto \sigma_1'(d_i)]_{1\le i\le k} \\
&\quad\ [\sigma_1'(v) = \sigma_1'(p_i^f)\ ?\ d_{i+k} \mapsto \texttt{tt} : d_{i+k} \mapsto \sigma_1'(d_{i+k})]_{1\le i\le w} \\
[\![call]\!]\sigma_1' &= \sigma_1'[g_i^f \mapsto g_i]_{1\le i\le k} \\
&\quad\ [p_i \mapsto \sigma_1'(a_i)]_{1\le i\le w} \\
&\quad\ [p_i^f \mapsto \sigma_1'(a_i)]_{1\le i\le w} \\
&\quad\ [l_i \mapsto 0]_{1\le i\le n} \\
&\quad\ [d_i \mapsto \texttt{ff}]_{1\le i\le k+w} \\
[\![ret]\!](\sigma_1', \sigma_2) &= \sigma_2[l_i \mapsto \mathfrak{L}(l_i, \sigma_1', \sigma_2)]_{1\le i\le n} \\
&\quad\ [p_i \mapsto \mathfrak{L}(p_i, \sigma_1', \sigma_2)]_{1\le i\le w} \\
&\quad\ [g_i \mapsto \mathfrak{G}(i, \sigma_1', \sigma_2)]_{1\le i\le k} \\
&\quad\ [\texttt{ret} \mapsto \mathfrak{R}(v, \sigma_1', \sigma_2)]
\end{aligned}
$$

$$
\mathfrak{L}(v, \sigma_1, \sigma_2) = \begin{cases} 0 & \text{if } \left(\begin{array}{l} \exists\ j\ .\ 1 \le j \le k \wedge \sigma_2(v) = \sigma_2(g_j) \wedge \sigma_1(d_j) = \texttt{tt} \vee \\ \exists\ j\ .\ k+1 \le j \le k+w \wedge \sigma_2(v) = \sigma_2(p_j) \wedge \sigma_1(d_j) = \texttt{tt} \end{array}\right) \\ \sigma_2(v) & \text{otherwise} \end{cases}
$$

$$
\mathfrak{G}(i, \sigma_1, \sigma_2) = \begin{cases} 0 & \text{if} & \sigma_1(g_i) = 0 \\ \mathfrak{G}(j, \sigma_1, \sigma_2) & \text{if} & j \text{ is the least index s.t. } \sigma_1(g_i) = \sigma_1(g_j) \\ \sigma_1(g_j^f) & \text{if} & j \text{ is the least index s.t. } \sigma_1(g_i) = \sigma_1(g_j^f) \\ \sigma_1(p_j^f) & \text{if} & j \text{ is the least index s.t. } \sigma_1(g_i) = \sigma_1(p_j^f) \\ \mathfrak{N}(g_i, \sigma_2) & \text{otherwise} & \end{cases}
$$

$$
\mathfrak{R}(v, \sigma_1, \sigma_2) = \begin{cases} \mathfrak{G}(i, \sigma_1', \sigma) & \text{if} & i \text{ is the least index s.t. } \sigma_1'(v) = \sigma_1'(g_i) \\ \sigma_1'(g_i^f) & \text{if} & i \text{ is the least index s.t. } \sigma_1'(v) = \sigma_1'(g_i^f) \\ \sigma_1'(p_i^f) & \text{if} & i \text{ is the least index s.t. } \sigma_1'(v) = \sigma_1'(p_i^f) \\ 0 & \text{if} & v = 0 \\ \mathfrak{N}(v, \sigma_2) & \text{otherwise} & \end{cases}
$$

Example 14. An example sequential run through the branching behavior of the flow graph in Figure 9 is shown below. In the run, the boxes represent the equivalence classes of variables, where the left-most box always represents the class of `null`, e.g., $\boxed{\texttt{P}, \texttt{P}^f \mid \texttt{c}}$ shows that variables `P` and $\texttt{P}^f$ are in the equivalence class of `null` and `c` is in a different one. In this example, we do not show the values of $\mathbb{D}$ variables for simplicity. Their values do not effect the execution.

We start from initial program state $\boxed{\texttt{P},\ \texttt{P}^f, \texttt{c}^f \mid \texttt{c}}$.

$$(\langle s_1, \boxed{\texttt{P}, \texttt{P}^f, \texttt{c}^f \mid \texttt{c}}, \boxed{\texttt{P}, \texttt{P}^f, \texttt{c}^f \mid \texttt{c}} \rangle, \epsilon)$$

$$\xrightarrow{\varepsilon} (\langle s_2, \boxed{\texttt{P}, \texttt{P}^f, \texttt{c}^f \mid \texttt{c}}, \boxed{\texttt{P}^f, \texttt{c}^f \mid \texttt{P}, \texttt{c}} \rangle, \epsilon)$$

$$\xrightarrow{\texttt{Container call Servlet(c)}} (\langle s_6, \boxed{\texttt{n} \mid \texttt{P}, \texttt{P}^f, \texttt{c}, \texttt{c}^f}, \boxed{\texttt{n} \mid \texttt{P}, \texttt{P}^f, \texttt{c}, \texttt{c}^f} \rangle, \langle s_3, \boxed{\texttt{P}^f, \texttt{c}^f \mid \texttt{P}, \texttt{c}} \rangle)$$

$$\xrightarrow{\varepsilon} (\langle s_{12}, \boxed{\texttt{n} \mid \texttt{P}, \texttt{P}^f, \texttt{c}, \texttt{c}^f}, \boxed{\mid \texttt{n}, \texttt{P}, \texttt{P}^f, \texttt{c}, \texttt{c}^f} \rangle, \langle s_3, \boxed{\texttt{P}^f, \texttt{c}^f \mid \texttt{P}, \texttt{c}} \rangle)$$

$$\xrightarrow{\texttt{Servlet call Container(n)}} (\langle s_1, \boxed{\mid \texttt{P}, \texttt{P}^f, \texttt{c}, \texttt{c}^f}, \boxed{\mid \texttt{P}, \texttt{P}^f, \texttt{c}, \texttt{c}^f} \rangle, \langle s_{13}, \boxed{\mid \texttt{n}, \texttt{P}, \texttt{P}^f, \texttt{c}, \texttt{c}^f} \rangle \cdot \langle s_3, \boxed{\texttt{P}^f, \texttt{c}^f \mid \texttt{P}, \texttt{c}} \rangle)$$

$$\xrightarrow{\varepsilon} (\langle s_4, \boxed{\mid \texttt{P}, \texttt{P}^f, \texttt{c}, \texttt{c}^f}, \boxed{\mid \texttt{P}, \texttt{P}^f, \texttt{c}, \texttt{c}^f} \rangle, \langle s_{13}, \boxed{\mid \texttt{n}, \texttt{P}, \texttt{P}^f, \texttt{c}, \texttt{c}^f} \rangle \cdot \langle s_3, \boxed{\texttt{P}^f, \texttt{c}^f \mid \texttt{P}, \texttt{c}} \rangle)$$

$$\xrightarrow{\texttt{del(c)}} (\langle s_5, \boxed{\texttt{P}, \texttt{c} \mid \texttt{P}^f, \texttt{c}^f}, \boxed{\texttt{P}, \texttt{c} \mid \texttt{P}^f, \texttt{c}^f} \rangle, \langle s_{13}, \boxed{\mid \texttt{n}, \texttt{P}, \texttt{P}^f, \texttt{c}, \texttt{c}^f} \rangle \cdot \langle s_3, \boxed{\texttt{P}^f, \texttt{c}^f \mid \texttt{P}, \texttt{c}} \rangle)$$

$$\xrightarrow{\texttt{Container ret Servlet}} (\langle s_{13}, \boxed{\texttt{n}, \texttt{P}, \texttt{c} \mid \texttt{P}^f, \texttt{c}^f}, \boxed{\texttt{n}, \texttt{P}, \texttt{c} \mid \texttt{P}^f, \texttt{c}^f} \rangle, \langle s_3, \boxed{\texttt{P}^f, \texttt{c}^f \mid \texttt{P}, \texttt{c}} \rangle)$$

$$\xrightarrow{\texttt{del(n)}} (\langle s_{14}, \boxed{\texttt{n}, \texttt{P}, \texttt{c} \mid \texttt{P}^f, \texttt{c}^f}, \boxed{\texttt{n}, \texttt{P}, \texttt{c} \mid \texttt{P}^f, \texttt{c}^f} \rangle, \langle s_3, \boxed{\texttt{P}^f, \texttt{c}^f \mid \texttt{P}, \texttt{c}} \rangle)$$

$$\xrightarrow{\texttt{Servlet ret Container}} (\langle s_3, \boxed{\texttt{P}, \texttt{P}^f, \texttt{c}, \texttt{c}^f}, \boxed{\texttt{P}, \texttt{P}^f, \texttt{c}, \texttt{c}^f} \rangle, \epsilon)$$

Consider transition $\xrightarrow{\texttt{Container ret Servlet}}$ and how it updates the program states. At the return state $\langle s_5, (\boxed{\texttt{P}, \texttt{c} \mid \texttt{P}^f, \texttt{c}^f}, \boxed{\texttt{P}, \texttt{c} \mid \texttt{P}^f, \texttt{c}^f}) \rangle$ the global variable `P` is in the equivalence class of `null`. Thus, aliases of `P` (i.e., variables in the equivalence class of `P`) in the state before the call to `Container` (i.e., the state at the top of the stack $\langle s_{12}, \boxed{\mid \texttt{n}, \texttt{P}, \texttt{P}^f, \texttt{c}, \texttt{c}^f} \rangle$) are moved to the equivalence class of `null` in the state after the return from `Container` ($\langle s_{12}, \boxed{\texttt{n}, \texttt{P}, \texttt{c} \mid \texttt{P}^f, \texttt{c}^f} \rangle$).

Also consider the second global property in Figure 1. The transition $\xrightarrow{\texttt{del n}}$ falsifies this property because `n` belongs to the equivalence class of `null` in the state exactly before deleting.

9.3 Logics

The structural and behavioral logics are also defined as instantiations of generic simulation logic (Definition 9) for the sets of assertions and actions defined above. In Example 1, we showed a structural and a behavioral property for the PoP instantiation of the generic simulation logic. Here, we specify properties expressed informally in Figure 8.

Example 15. The local structural property of method `Servlet()` in Figure 8 is specified by the structural formula $(X_0 \wedge X_1 \wedge X_2)[\Pi]$, where Π is

$$\begin{aligned}
X_0 &= [\texttt{new(P)}]\texttt{ff} \wedge \bigwedge\nolimits_{l \in L_{\texttt{Serv}} \setminus \texttt{new(P)}} [l] X_0 \\
X_1 &= (\neg \texttt{r} \wedge \bigwedge\nolimits_{l \in L_{\texttt{Serv}} \setminus \{\texttt{del(c)}\}} [l] X_1) \vee \texttt{n'=c} \\
X_2 &= [\texttt{del(n)}] X_3 \wedge \bigwedge\nolimits_{l \in L_{\texttt{Serv}} \setminus \{\texttt{del(n)}\}} [l] X_2 \\
X_3 &= \bigwedge\nolimits_{l \in L_{\texttt{Serv}}} [l]\texttt{ff} \wedge \texttt{n'=n}
\end{aligned}$$

where `Container` is the set of labels representing calls to the method `Container`. In the formula, X_0 formalizes "no `new P` statement", the equation X_1 formalizes "`n := c` or `del c` before return", and the equations X_2 and X_3 together formalize "no statement after `del n`". Notice that the formula does not formalize the

constraint "no assignment statement to `P`", because in the interface of method `Servlet` (given in the Flow Graph paragraph), $\texttt{P} \notin \mathit{Modify}$.

The first global behavioral property in Figure 8 is specified by the behavioral formula $X[\Pi]$, where Π is

$$X = \bigwedge_{l \in L_b} [l]X \ \wedge \ (\texttt{r} \wedge \texttt{Container} \rightarrow (\texttt{c} = \texttt{null}))$$

The syntax of the formulas may look complicated, but syntactic sugar can be introduced by means of macros or specification templates to simplify the presentation (e.g., see [19]).

9.4 Maximal Flow Graph

The maximal flow graphs are constructed by applying the algorithm described in Section 5 for the set of structural labels (method names and actions), atomic propositions (method names and `r`), and the top element of the lattice ordered by $\sqsubseteq$ (which is here logical implication).

Example 16. We construct the maximal flow graph of method `Servlet` from its interface and local structural property. To do this, the following characteristic formula is constructed from the interface.

$(X0 \vee X1)[\Pi]$ where Π is

$$\begin{aligned} X0 = {} & ([\varepsilon](X0 \vee X1) \wedge [\texttt{Container}](X0 \vee X1) \wedge \\ & [\texttt{Servlet}](X0 \vee X1) \wedge \texttt{r} \wedge \texttt{Servlet} \wedge P' = P) \\ X1 = {} & ([\varepsilon](X0 \vee X1) \wedge [\texttt{Container}](X0 \vee X1) \wedge \\ & [\texttt{Servlet}](X0 \vee X1) \wedge \neg\texttt{r} \wedge \texttt{Servlet} \wedge P' = P) \end{aligned}$$

The above formula is conjoined with the local structural property of `Servlet` and maximal flow graph shown in Figure 10 is constructed.

9.5 Compositional Verification

We use Rule 2 for the program models, logics and maximal flow graphs defined for PoP programs. As a result, we obtain a compositional verification technique for PoP programs.

Example 17. Consider the program in Figure 8 and its properties. We use the PoP instantiation of Rule 2 to verify this program, assuming method `Servlet` as a variable component. This divides the verification into the following two independent sub-tasks. (i) We check that the local specification of method `Servlet` is satisfied by its implementation (whenever it is available or changed). The implementation of `Servlet` in the figure satisfies its local structural specification. (ii) A maximal flow graph is constructed from the interface and local structural property of method `Servlet` and composed with the flow graph of method `Container`. Then the behavior induced from the resulting flow graphis model checked against the global properties of the program.

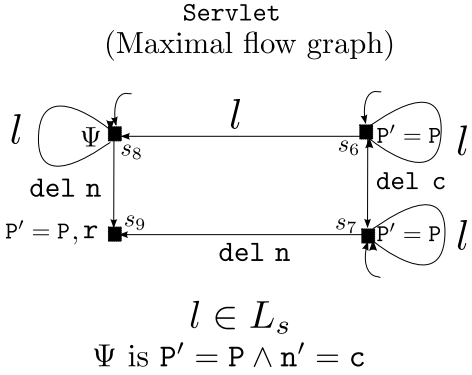


Fig. 10: Maximal flow graph constructed from local structural property and interface of method `Servlet`

9.6 Tool Support and Evaluation.

We have extended our compositional verification toolset [22] for the verification of PoP programs in the presence of variability. Apart from 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 [14] and Moped [24]. 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 [6,17]. 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.

10 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 [18], and later generalized by Kupferman and Vardi for ACTL* [26]. 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 [19]. 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. [28]) 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) [7]. 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 [12] and ESP [16] 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 [5] and its generalization to nested words [4,1] 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.

11 Conclusion

This paper presents a generic framework for compositional verification of temporal safety properties of sequential procedural programs in the presence of vari-

ability. 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 *Logic in Computer Science (LICS '07)*, pages 151–160, Washington, DC, USA, 2007. IEEE Computer Society.
2. R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27:786–818, 2005.
3. R. Alur and S. Chaudhuri. Temporal reasoning for procedural programs. In *Verification, Model Checking, and Abstract Interpretation (VMCAI '10)*, volume 5944 of *Lecture Notes in Computer Science*, pages 45–60. Springer, 2010.
4. R. Alur, S. Chaudhuri, and P. Madhusudan. Languages of nested trees. In *Computer Aided Verification (CAV '06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 329–342. Springer, 2006.
5. 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 *Lecture Notes in Computer Science*, pages 467–481. Springer, 2004.
6. A. Amighi, P. de Carvalho Gomes, D. Gurov, and M. Huisman. Sound control-flow graph extraction for java programs with exceptions. In G. Eleftherakis, M. Hinchey, and M. Holcombe, editors, *Software Engineering and Formal Methods (SEFM)*, volume 7504 of *Lecture Notes in Computer Science*, pages 33–47. Springer Berlin / Heidelberg, 2012.
7. H. Andersen. Partial model checking (extended abstract). In *Logic in Computer Science (LICS '95)*, pages 398–407. IEEE Computer Society Press, 1995.

8. A. Arnold and D. Niwiński. *Rudiments of μ-calculus*, volume 146 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Publishing, 2001.
9. T. Ball, A. Podelski, and S. Rajamani. Boolean and cartesian abstraction for model checking c programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 268–283. Springer, 2001.
10. T. Ball and S. Rajamani. Boolean programs: A model and process for software analysis. Technical Report MSR-TR-2000-14, Microsoft Research, 2000.
11. T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 113–130, London, UK, UK, 2000. Springer-Verlag.
12. T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Principles of programming languages (POPL '02)*, pages 1–3, 2002.
13. H. Bekič. Definable operators in general algebras, and the theory of automata and flowcharts. Technical report, IBM Laboratory, 1967.
14. 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.
15. B. Cook, D. Kroening, and N. Sharygina. *Symbolic model checking for asynchronous boolean programs*. Springer, 2005.
16. M. Das, S. Lerner, and M. Seigle. ESP: Path–sensitive program verification in polynomial time. In *Programming Language Design and Implementation (PLDI '02)*, pages 57–68. ACM, 2002.
17. P. de Carvalho Gomes, A. Picoco, and D. Gurov. Sound control flow graph extraction from incomplete java bytecode programs. In *Fundamental Approaches to Software Engineering (FASE)*, volume 8411 of *Lecture Notes in Computer Science*, pages 215–229, Berlin, 2014. Springer.
18. O. Grumberg and D. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):843–871, 1994.
19. D. Gurov, M. Huisman, and C. Sprenger. Compositional verification of sequential programs with procedures. *Information and Computation*, 206(7):840–868, 2008.
20. M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32:137–161, 1985.
21. M. Huisman, I. Aktug, and D. Gurov. Program models for compositional verification. In *International Conference on Formal Engineering Methods (ICFEM '08)*, volume 5256 of *Lecture Notes in Computer Science*, pages 147–166. Springer, 2008.
22. M. Huisman and D. Gurov. CVPP: A tool set for compositonal verification of control-flow safety properties. In *Formal Verification of Object–Oriented Software (FoVeOOS '10)*, volume 6528 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2010.
23. Servlet Development. Release 2 (9.0.3) `http://docs.oracle.com/cd/A97688_16/generic.903/a97680/develop.htm#1007089`.
24. S. Kiefer, S. Schwoon, and D. Suwimonteerabuth. Moped - a model-checker for pushdown systems. `http://www.informatik.uni-stuttgart.de/fmi/szs/tools/moped/`.
25. D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

26. O. Kupferman and M. Vardi. An automata-theoretic approach to modular model checking. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):87–128, 2000.
27. K. Larsen. Modal specifications. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 232–246. Springer, 1989.
28. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer, 2002.
29. A. Podelski and T. Wies. Boolean heaps. In C. Hankin and I. Siveroni, editors, *Static Analysis*, volume 3672 of *Lecture Notes in Computer Science*, pages 268–283. Springer Berlin Heidelberg, 2005.
30. J. Rot, F. de Boer, and M. Bonsangue. A pushdown system representation for unbounded object creation. In *Informal pre-proceedings of Formal Verification of Object–Oriented Software (FoVeOOS '10)*, 2010.
31. J. Rot, F. S. de Boer, and M. M. Bonsangue. Unbounded allocation in bounded heaps. In *Fundamentals of Software Engineering (FSEN)*, volume 8161 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2013.
32. 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.
33. S. Soleimanifard, D. Gurov, and M. Huisman. Procedure-modular specification and verification of temporal safety properties. *Software & Systems Modeling*, pages 1–18, 2013.
34. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–310, 1955.