Information and Computation 206 (2008) 840–868

Contents lists available at ScienceDirect

Information and Computation

journal homepage: www.elsevier.com/locate/ic

Compositional verification of sequential programs with procedures

Dilian Gurov [a,*], Marieke Huisman [b,1], Christoph Sprenger [c,2]

[a] Royal Institute of Technology, Department of Theoretical Computer Science, SE-100 44 Stockholm, Sweden
[b] INRIA, Sophia Antipolis, 2004, route des Lucioles, FR-06902 Sophia Antipolis, France
[c] ETH, Zurich, Chair of Information Security, CH-8092 Zurich, Switzerland

ARTICLE INFO

Article history:
Received 24 August 2006
Accepted 19 December 2007
Available online 9 May 2008

Keywords:
Program verification
Control-flow behaviour
Compositional reasoning
Modal μ-calculus
Safety properties
Maximal model
Private procedures

ABSTRACT

We present a method for algorithmic, compositional verification of control-flow-based safety properties of sequential programs with procedures. The application of the method involves three steps: (1) decomposing the desired global property into local properties of the components, (2) proving the correctness of the property decomposition by using a maximal model construction, and (3) verifying that the component implementations obey their local specifications. We consider safety properties of both the structure and the behaviour of program control flow. Our compositional verification method builds on a technique proposed by Grumberg and Long that uses maximal models to reduce compositional verification of finite-state parallel processes to standard model checking. We present a novel maximal model construction for the fragment of the modal μ-calculus with boxes and greatest fixed points only, and adapt it to control-flow graphs modelling components described in a sequential procedural language. We extend our verification method to programs with private procedures by defining an abstraction, presented as an inlining transformation. All algorithms have been implemented in a tool set automating all required verification steps. We validate our approach on an electronic purse case study.

1. Introduction

Motivation. Over the last years, computer systems have become increasingly dynamic: they are composed of various communicating components that can join the system or be put together dynamically. Typical examples are mobile smart devices (mobile phones, smart cards, television set top boxes, PDAs, etc.) and dynamically reconfiguring distributed systems. When allowing the dynamic addition of new components, one wishes to ensure that this will not have any negative impact on the global behaviour of the system. In particular when the system contains privacy-sensitive information, as is for example the case for smart cards containing health care information or electronic purses, strong security guarantees are required. With the acceptance of evaluation schemes such as Common Criteria (see [1]), industry has come to realise that the way to achieve such high guarantees is to adopt the use of formal methods in industrial practice.

The techniques developed here are applicable in any context concerned with interprocedural control-flow properties of components communicating via procedure calls. Interesting properties of such components include for example type safety, memory consumption, and illicit data or control flow. Here we concentrate on the last category of properties. More precisely, we study sequential (i.e., single-threaded) programs and propose a specification and verification method for safety properties

* Corresponding author. Fax: +46 8 790 09 30. This author's work was partially funded by the SEFROS project of the Swedish Research Council VR, and by the IST FP6 programme of the EC, under the IST-FP6-STREP-27004 S3MS project.
E-mail addresses: dilian@csc.kth.se (D. Gurov), marieke.huisman@sophia.inria.fr (M. Huisman), christoph.sprenger@inf.ethz.ch (C. Sprenger).
[1] This work was funded in part by the IST programme of the EC, FET under the IST-2005-015905 MOBIUS project.
[2] This work was partially supported by the Zurich Information Security Center. It represents the views of the authors.

0890-5401/$ - see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.ic.2008.03.003

of inter-procedural control flow, i.e., properties describing safe sequences of procedure invocations. Typical examples of control-flow safety properties are: "m_1 never calls m_2", "m_1 is never called when m_2 is called", "m_1 is only called after m_2 is called", and "m_1 is only called from within m_2" (see Chugunov et al. [2] for a formalisation).

So far, most research on formal verification in this area has focused on the correctness or security of a single program component (e.g., [3,4,5]). However, in the context of mobile code we also need techniques to support verification of systems for which it is not known in advance what its components will be. In such situations one needs *compositional verification techniques*, that is techniques where one states minimal requirements for the components that can become available later, and then verifies (at loading time) that the components actually respect these requirements. Only then, existing components can safely communicate with new components, without corrupting the correctness or security of the whole system. In particular, such techniques can support the secure post-issuance loading of new applications onto smart devices. To avoid false negatives, i.e., rejecting components that are actually secure, such compositional verification techniques should not only be sound, but also complete. Completeness is also crucial to avoid typical social engineering attacks, where the device user gets so frustrated with the system repeatedly rejecting new components, that he/she will simply accept all, without actually inspecting whether they passed verification or not.

Approach. Our verification method is *compositional*: it allows global guarantees of a system to be verified even if the implementations of some components are not yet available at verification time. This is achieved by abstracting the missing components by logical assumptions. These assumptions can be verified later, when the implementations become available. Such a verification approach is embodied by the following proof principle:

$$\frac{\vdash A : \phi \qquad X : \phi \vdash X \otimes B : \psi}{\vdash A \otimes B : \psi}$$

where A and B are components, and X is a component variable. This principle reduces the problem of showing that the composition of components A and B satisfies ψ, where the implementation of A is not yet known, to three tasks:

(1) decompose the global property ψ by finding a suitable local property ϕ of component A,
(2) prove *correctness* of the decomposition, i.e., verify that for *any* component X satisfying ϕ, X composed with B satisfies ψ (second premise), and
(3) when the implementation of A becomes available, verify that it satisfies the local property ϕ (first premise).

Notice that this rule can be applied repeatedly, to replace several components by assumptions.

The compositionality of the method supports different scenarios for secure configuration of components on a device (or platform), where the tasks above can potentially be delegated to different authorities. In one such scenario, the device issuer (or platform provider) specifies both the global guarantee (e.g., a security policy) and the local assumptions, and verifies—using the techniques described in this paper—that the decomposition is correct, meaning that the local specification is sufficient to establish the global specification. Each time a new component is to be added (i.e., loaded on the device), an algorithm provided by the device issuer checks whether the component implementation satisfies the required specification. An alternative scenario is that the device issuer only provides the global guarantee (and local assumptions for its own components), and leaves it to the component provider to come up with an appropriate local specification for each component to be added. As in the previous scenario, an algorithm provided by the device issuer checks the component against the local specification upon loading, but now also the property decomposition needs to be verified at loading time, potentially on-device.

Task (1) above is a manual one and requires insight into the system, while the other two can be automated in our approach. We show how Tasks (2) and (3) can be algorithmically reduced to problems for which standard algorithmic techniques exist.

The approach that we take to handle Task (2) is inspired by the pioneering work on automatic modular verification by Grumberg and Long [6]. To check whether $X : \phi \models X \otimes B : \psi$ holds we replace X by a *maximal model* $\theta(\phi)$ and then verify $\models \theta(\phi) \otimes B : \psi$ algorithmically. The maximal model $\theta(\phi)$ represents all models satisfying ϕ in the sense that it *simulates* exactly those models and thus satisfies precisely the properties enjoyed by all these models. For this technique to be sound and applicable it is required that maximal models exist for the chosen logic and simulation relation, $\otimes$ preserves simulation, and logical properties are preserved by simulation. In earlier work [7], we explored deductive verification of correctness of decompositions based on a proof system. The logic considered there was more expressive, but the interactive nature of the approach required considerable time and expertise from the user, rendering the approach less preferable in many situations as compared to algorithmic solutions like the one presented here.

We are interested in *safety properties* of both the *structure* and the *behaviour* of programs. Since the same behaviour can be brought about by different structures, a behavioural property language allows properties to be expressed in a more abstract fashion. However, as a rule, behavioural properties require computationally more expensive verification techniques. Still, they can often be (equivalently) reformulated on the structural level, with the advantage of allowing more efficient verification. To support both kinds of properties, we distinguish between a structural and a behavioural level of programs. Both structure and behaviour are cast via the abstract notion of *model* (or labelled Kripke structure). Then, structural properties are interpreted over the (finite-state) control-flow graphs themselves, while behavioural properties are interpreted over the (infinite-state) behaviours induced by the structures. The logic we employ to express such properties is a modal logic with box modalities and simultaneous greatest fixed points (written in equational form), which is expressively equivalent to the fragment of the

modal μ-calculus with box modalities and greatest fixed points only [8]. The fragment is known to be adequate for expressing safety properties (cf. [9]). Because of the close relationship between logical satisfaction and *simulation* between models, and the compositional properties of simulation, this logic, which for convenience we term *simulation logic*, is particularly suitable for compositional verification via maximal models. We instantiate simulation logic and simulation at both the structural and the behavioural levels.

The methods provided by an applet are frequently implemented using internal, private methods. Since the private methods cannot be expected to be known before the applet is implemented, we introduce *public interfaces*, which hide private methods. Accordingly, the *(public) interface behaviour* of an applet abstracts from (internal) calls to the private methods of an applet. To handle Task (3) for programs with private procedures, we define an *inlining transformation* that recursively inlines all calls to private procedures. This transformation over-approximates the interface behaviour, and reduces the task to showing that the inlined program respects property ϕ. For the latter, we apply standard algorithmic verification techniques.

Contributions. The main contribution of the present paper is a sound and complete compositional verification principle for sequential programs with procedures, for properties expressed in simulation logic, and its adaptation to programs with private procedures. In more detail, the contributions are as follows.

(1) *Program model.* Most of the existing work on compositional model checking focuses on the verification of parallel compositions of finite-state processes. We extend compositional model checking to an important class of infinite-state programs, namely sequential programs with procedures. In the rest of this paper, we refer to programs as *applets* and to procedures as *methods*, but we would like to stress that our technique is applicable to many different kinds of programs with procedures. We represent applets as collections of method control-flow graphs equipped with *interfaces* of provided and required methods. Applet *composition* forms the disjoint union of the respective collections of method graphs and allows the composed applets to communicate via method invocation. Applets correspond to a subclass of pushdown processes, with potentially infinite-state behaviour (cf. Burkart et al. [10]).

(2) *Maximal model Construction.* We establish a *logical characterisation* of the standard notion of simulation between models and, vice versa, a behavioural characterisation of logical satisfaction in terms of *maximal models*. In particular, we present a novel maximal model construction, consisting of a step-wise transformation of the formula into a semantically equivalent normal form, which is isomorphic to a maximal model for the formula. In contrast to more expressive logics, the maximal models for simulation logic formulae are representable as standard transition systems. To the best of our knowledge, this is the first maximal model construction for (a variant of) the modal μ-calculus, which includes the full expressive power of simultaneous greatest fixed points.

(3) *Maximal applet construction.* When tailoring the maximal model technique to applets, we require that the maximal model for a given property is itself an applet. This is necessary for *completeness* of the technique. Since the verification of $\models \theta(\phi) \otimes B : \psi$ is decidable in our setup, completeness guarantees that if the verification of the correctness of decomposition fails, there is indeed an *applet* F among the set of models such that F satisfies ϕ but $F \otimes B$ does not satisfy ψ. Completeness is thus essential in that it eliminates the possibility of false negatives. Therefore, in case $\models \theta(\phi) \otimes B : \psi$ fails, we know that we have to strengthen ϕ and iterate the process.
To adapt the maximal model technique to structural properties, we first give a logical characterisation of interfaces by defining, for a given interface I a structural formula ϕ_I which is satisfied exactly by those models representing applet structures with this interface, and then define the *maximal applet* for a given interface I and structural property ϕ by $\theta_I(\phi) = \theta(\phi_I \wedge \phi)$. Since $\theta(\phi_I \wedge \phi)$ satisfies both ϕ and ϕ_I, this guarantees that the resulting maximal model is indeed an applet structure with interface I satisfying the structural formula ϕ.
However, for behavioural properties there is in general no unique maximal applet: different applets, incomparable by simulation, might exist that satisfy the same property. It is ongoing work to investigate under what conditions and how this collection of maximal applets can be characterised exactly. Preliminary results in this direction are presented by Gurov and Huisman in [11].

(4) *Compositional verification.* Our characterisation results, together with results linking the structural and behavioural levels, give rise to a *compositional verification principle* of the shape suggested above, where the global guarantee can be either structural or behavioural, but the local assumptions are always structural. We establish the *soundness* and *completeness* of the principle, and adapt existing algorithmic techniques for dealing with the resulting verification sub-tasks.

(5) *Interface abstraction.* We extend our compositional verification method to *interface properties* of applets, i.e., properties of the interface behaviour. We define an *abstraction* which reduces the set of methods of a given applet to the set of its public methods, while over-approximating the interface behaviour of the applet. This abstraction is based on *inlining* of private methods. We show the abstraction to be *sound* with respect to interface properties: every interface property that holds for the behaviour of the inlined applet also holds for the interface behaviour of the original applet. Since the abstraction transformation may introduce new interface behaviours, completeness, on the other hand, does not hold in general. However, for the case when the concrete implementation is *last-call recursive* (that is, recursive calls are not followed in the control-flow graph by any other method calls), the abstraction technique is *complete* with respect to observable interface properties: if such a property does not hold of the inlined applet it does not hold of the original applet either. Last-call recursion is a generalisation of the notion of *tail recursion*, where recursive calls are the last statements of their methods. In practice, for industrial code it is very common to be last-call recursive.

(6) *Tool support and real-life case study.* To support our compositional verification technique, we have developed a tool set. This tool set integrates our own implementations in Ocaml of the maximal applet construction and the inlining algorithm with an implementation of a model extractor, build on top of the SOOT framework [12], and a number of external model checking tools. We have validated this tool set on an industrial case study, namely an electronic purse smart card applet for which we have verified the absence of certain illicit control flows between *Purse* and *Loyalty* applets. In particular, we ensured that different *Loyalty* applets on the card cannot communicate information about the transaction log table – that is needed to correctly compute the points in the loyalty program – among themselves, instead they all need to register (and pay) to get this information directly from the *Purse*. In this case study, the inlining technique proved to be an essential ingredient that enabled the compositional verification of the otherwise too large model.

Our contributions span the complete spectrum from the theoretical underpinnings of the compositional applet verification technique (our principal contribution) to its support by a tool set and its application to an industrial case study.

Related work. The work presented here is related to several different research areas.

Program model. The program model used in the present paper has been inspired by the work of Besson et al. [4], who verify stack properties for Java programs. Typically, the behaviour of programs with recursion is modelled as Pushdown Automata (as, e.g., in [3,13]).

Recursive state machines were introduced by Alur et al. [5] as a formalism capable of modelling the control flow of sequential imperative programs containing recursive procedure calls. This program model is closely related to our own, but is finer in that calls and returns relate individual entry and return nodes, thus allowing the effect of data to be modelled. The authors develop efficient algorithms for (global) model checking of recursive state machines against LTL and CTL* properties, and investigate their complexity.

Temporal logic. Related to the above program models is the temporal logic of calls and returns CaRet proposed by Alur et al. [14]. This logic allows to specify properties in terms of method calls and returns, thus increasing the expressiveness of temporal logic while retaining decidability of model checking. A special verification strategy is defined, that is able to "jump over" internal computations. An extension of this logic was recently presented by Alur et al. [15]. Among other modalities, it introduces the useful "within" modality, which is not expressible in simulation logic. While these logics may be more adequate than simulation logic for specifying behavioural properties of programs with procedures, they would (arguably) require more involved techniques for compositional verification.

Compositional verification. There is a wealth of methods for compositional verification of concurrent programs, most notably assumption/commitment based reasoning about processes with synchronous message passing, and the rely/guarantee method for shared-variable concurrency. A systematic overview of these and related proof methods, some of which have been adapted to support algorithmic verification is given by De Roever et al. [16]. However, these techniques do not address programs with recursive procedures.

Laster and Grumberg [17] present a compositional method for sequential programs written in a high-level While language (without procedures). Their technique partitions the program text into a sequence of sequentially composed subprograms, which can be model checked individually using assumptions on the properties holding at the cut points.

Alur and Grosu [18] present an assume-guarantee style compositional verification principle for a hierarchic extension of reactive state machines. However, their approach does not address programs with recursion.

Ly [19] also proposes a compositional method for deciding control-flow properties of procedural programs based on local structural assumptions and global behavioural guarantees. The author generalises our decidability results to monadic second-order logic for programs whose control-flow graphs have a bounded tree-width. To the best of our knowledge, so far this approach has not been implemented in a tool.

The method of partial model checking introduced by Andersen [20] is based on a reduction procedure that removes the top-level operator from a process algebra term and computes a new property for the reduced term. To verify that the product $P \times Q$ of two processes has some property ϕ, the reduction "divides" the property ϕ by Q to yield ϕ/Q, which can be effectively computed only if Q is finite.

Maximal models for compositional verification. The original maximal model technique by Grumberg and Long [6] was designed for ACTL, the universal fragment of CTL, and later extended to ACTL*, the universal fragment of CTL*, by Kupferman and Vardi [21]. These works study synchronous parallel compositions of sequential processes under fairness assumptions. Since we are interested in safety properties of sequential programs, we do not need to add fairness to our models. Simulation logic and ACTL* are expressively incomparable: liveness properties such as *GFp* ("infinitely often p") are expressible in ACTL*, but not in simulation logic, while the μ-calculus formula $\nu X.\, p \wedge [-][-]X$ ("p holds on every other level of the computation tree") is easily translated to simulation logic (which is in equational form), but is not expressible in ACTL*. Our transformational approach to the maximal model construction is closer to an implementation than the automata-theoretic constructions in the cited papers, since it already includes certain optimisations, e.g., removal of duplicate and unreachable equations.

Characterisation results connecting logics and behavioural preorders similar to ours are described by Boudol and Larsen [22] (see also [23]), who construct maximal models in the form of modal transition systems with respect to the refinement preorder for Hennessy–Milner logic (HML) [24]. Simulation logic and HML are expressively incomparable: existential properties are not expressible in simulation logic, while co-recursive properties (such as invariants) are not expressible in HML. Since HML does not include fixed points, the constructed maximal models are essentially finite forests. Apart from the absence of

diamond modalities in simulation logic, our construction can be seen as an extension of Larsen and Boudol's with greatest fixed points. The extension of HML with greatest fixed points (or, equivalently, simulation logic with diamond modalities) requires more general models than modal transition systems: a finite maximal modal transition system does not exist for all formulae of this logic. This is shown by Dams and Namjoshi [25], who introduce focus transition systems, generalising modal transition systems, in order to construct linear-size maximal models for properties expressed by alternating tree automata (thus subsuming the full modal μ-calculus). In [26] the same authors propose to directly use μ-automata obtained from modal μ-calculus formulae as maximal models, for which they define an appropriate notion of simulation. All natural extensions of simulation logic require models with more structure than transition systems to capture maximal models. In our work, we were interested in safety properties, for which simulation logic and transition systems are an appropriate choice.

Bouajjani et al. [9] define maximal models for a co-recursive modal logic expressing safety properties. Their logic has an expressive power similar to ours, but is somewhat less standard as it includes a connective corresponding to non-deterministic choice.

A more recent application of the maximal model technique is presented by Goldman and Katz [27] in the context of modular verification of *aspects*. While close in spirit to our verification principle, the principle presented by the authors is for a more complicated composition operator. The principle is based on the maximal model of the aspect property (which is not necessary a legal aspect behaviour) and is therefore sound, but not complete.

Organisation. The paper is structured as follows. First, Section 2 presents the theoretical foundation for our work: it defines the models and logic that we consider, together with appropriate notions of simulation and satisfaction. Next, Section 3 presents our novel maximal model construction, and shows how logical satisfaction of a formula is equivalent to simulation by the corresponding maximal model. Section 4 then discusses how our results instantiate to applets, at structural and at behavioural level, and Section 5 presents the compositional verification principle. Section 6 presents the inlining abstraction that we use to be able to verify interface properties over applets with private methods. Finally, Sections 7 and 8 illustrate how our approach is implemented as a tool set and is applied to an industrial case study, while Section 9 draws conclusions and presents future work.

This paper is a combination and extension of several results presented earlier. The maximal model construction and compositional verification principle are presented in [28]. The abstraction technique for applets with private methods is presented in [29]. The case study was presented in [30], but without taking the difference between public and private methods into account.

2. Models, simulation and logic

This section describes the theoretical foundation for our treatment of control-flow structure and behaviour of programs with recursive procedures. First, we define the (abstract) models that we study, together with the standard notion of simulation. Further, we define the logic that we use to express our program properties. Finally, we transfer all these notions to the so-called weak setting, where not all actions are observable.

2.1. Model and simulation

First we define models, specifications and simulation. These notions are standard up to some minor variations.

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

The reachable part of a specification $\mathcal{S} = (\mathcal{M}, E)$ is defined by $\mathcal{R}(\mathcal{S}) = (\mathcal{M}', E)$, where $\mathcal{M}'$ is obtained from $\mathcal{M}$ by deleting all states and transitions not reachable from any entry state in E.

Example 2. Fig. 1 shows the graphical representation of the specification $\mathcal{S} = (\mathcal{M}, E)$, where $\mathcal{M} = (\{s_1, s_2, s_3\}, \{a, \varepsilon\}, \rightarrow, \{p, q\}, \{s_1 \mapsto \{p, q\}, s_2 \mapsto \{p\}, s_3 \mapsto \varnothing\})$ with $\rightarrow = \{(s_1, \varepsilon, s_2), (s_2, a, s_1), (s_2, a, s_3), (s_3, a, s_1), (s_3, \varepsilon, s_2)\}$ and $E = \{s_1, s_2\}$. As usual, entry states are depicted through additional incoming edges without source.

Definition 3 (*Simulation*)*. A simulation is a binary relation R on S such that whenever $(s, t) \in R$ then $\lambda(s) = \lambda(t)$, 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

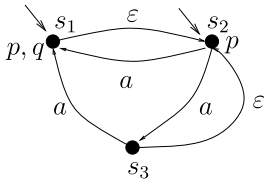

Fig. 1. Example specification $\mathcal{S} = (\mathcal{M}, E)$.

S_i into $S_1 \uplus S_2$. Simulation is extended to specifications $(\mathcal{M}_1, E_1)$ and $(\mathcal{M}_2, E_2)$ 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 $(in_1(s), in_2(t)) \in R$. Specification $\mathcal{S}_1$ 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 specifications (by $(\mathcal{M}_1, E_1) \uplus (\mathcal{M}_2, E_2) = (\mathcal{M}_1 \uplus \mathcal{M}_2, E_1 \uplus E_2)$) and show that simulation is preserved by disjoint union.

Theorem 4. *If $\mathcal{S}_1 \leqslant \mathcal{T}_1$ and $\mathcal{S}_2 \leqslant \mathcal{T}_2$ then $\mathcal{S}_1 \uplus \mathcal{S}_2 \leqslant \mathcal{T}_1 \uplus \mathcal{T}_2$.*

2.2. Simulation logic

We define simulation logic in two steps: first we define a basic modal logic, and then we add recursion by means of equation systems. This results in a logic that is equally expressive as the modal μ-calculus with boxes and greatest fixed points only (cf. Bekič [31]). However, the use of equation systems facilitates the definition of a normal form, where the correspondence between formulae and specifications is immediate. In particular, this allows to compute maximal models by transforming the equations into this normal form.

Let $\mathcal{V}$ be a countably infinite set of propositional variables. *Basic simulation logic* is a variant of Hennessy–Milner logic [24] without diamond modalities:

$$\phi ::= \mathsf{ff} \mid \mathsf{tt} \mid p \mid \neg p \mid X \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid [a]\,\phi$$

where $p \in A$, $a \in L$ and $X \in \mathcal{V}$. The interpretation $\|\phi\|\rho$ of a basic formula ϕ is defined with respect to a model $\mathcal{M}$ and an environment ρ interpreting the propositional variables. The definition is standard (cf. Stirling [32]); in particular, for the box modality we have $s \in \|[a]\,\phi\|\rho$ if and only if for all $t \in S$ such that $s \xrightarrow{a} t$ we have $t \in \|\phi\|\rho$. Formulae like p or $\neg p$ are called *literals*. We use n-ary versions of conjunction and disjunction, setting $\bigvee \varnothing = \mathsf{ff}$ (false) and $\bigwedge \varnothing = \mathsf{tt}$ (true). As usual, for finite $K \subseteq L$, we write $[K]\,\phi$ for $\bigwedge_{a \in K} [a]\,\phi$ and $[-]\,\phi$ for $[L]\,\phi$.

To make the logic expressive enough to characterise all finite models, we follow Larsen [23] and add recursion to basic simulation logic by introducing modal equation systems. A *modal equation system* Σ is a finite set of defining equations of the shape $X = \phi_X$, where X is a propositional variable and ϕ_X is a formula of basic simulation logic. The defined variables X are pairwise distinct and bound in Σ, while all other variables are free. For a simpler presentation, we restrict our attention here to closed equation systems without free variables.

Since the considered equations systems are closed, it is sufficient to work with environments $\rho : \mathsf{bv}(\Sigma) \to \mathcal{P}(S)$ mapping the bound variables of Σ to sets of states. The equations in Σ induce a map $\Psi_\Sigma : \mathcal{P}(S)^{\mathsf{bv}(\Sigma)} \to \mathcal{P}(S)^{\mathsf{bv}(\Sigma)}$ on such environments ρ defined by $\Psi_\Sigma(\rho)(X) = \|\phi_X\|\rho$. A solution of Σ is an environment ρ such that all equations in Σ are satisfied (that is, $\Psi_\Sigma(\rho) = \rho$), and is thus a *fixed point* of Ψ_Σ. Environments are ordered by point-wise inclusion. The *semantics* of a modal equation system Σ with respect to a model $\mathcal{M}$, denoted $\|\Sigma\|$, is its greatest solution. By the Knaster-Tarski fixed point theorem [33] a greatest solution always exists, since Ψ_Σ is a monotone function on the complete lattice of environments ordered by point-wise set inclusion.

Definition 5 (*Simulation logic*)**.** A (closed) formula of *simulation logic* has the shape $\phi[\Sigma]$, where ϕ is a formula of basic simulation logic and Σ is a (closed) modal equation system such that all variables occurring in ϕ are bound in Σ. The *semantics of* $\phi[\Sigma]$ with respect to model $\mathcal{M}$ is defined by $\|\phi[\Sigma]\| = \|\phi\|\|\Sigma\|$. We say a specification $(\mathcal{M}, E)$ satisfies $\phi[\Sigma]$, written $(\mathcal{M}, E) \models \phi[\Sigma]$, if $E \subseteq \|\phi[\Sigma]\|$.

Example 6. Consider the formula $\phi = (X \vee Y)[\Sigma]$, where

$$\Sigma = \begin{bmatrix} X & = & [\varepsilon]\,Y \wedge [a]X \wedge p \\ Y & = & [\varepsilon]\,(X \wedge Y) \wedge \neg q \end{bmatrix}.$$

Let us determine the semantics of this formula with respect to the specification $\mathcal{S}$ in Fig. 1. The greatest fixed point $\|\Sigma\|$ of Ψ_Σ with respect to $\mathcal{S}$ can be computed in the standard way by iteration of Ψ_Σ starting with $\rho_0 = \{X \mapsto S, Y \mapsto S\}$, where $S = \{s_1, s_2, s_3\}$. This yields $\|\Sigma\| = \{X \mapsto \{s_1\}, Y \mapsto \{s_2\}\}$. So, $E = \|X \vee Y\|\|\Sigma\| = \{s_1, s_2\}$, and hence specification $\mathcal{S}$ satisfies ϕ.

Henceforth, we often omit the equation system Σ from $\phi[\Sigma]$ if no confusion can arise. We say that ϕ_1 is a logical consequence of ϕ_0, written $\phi_0 \sqsubseteq \phi_1$, if for all specifications $\mathcal{S}$, $\mathcal{S} \models \phi_0$ implies $\mathcal{S} \models \phi_1$. The formula ϕ_0 is logically equivalent to ϕ_1, written $\phi_0 \equiv \phi_1$, if $\phi_0 \sqsubseteq \phi_1$ and $\phi_1 \sqsubseteq \phi_0$.

Simulation logic is equally expressive as the modal μ-calculus [8] without diamond modalities and least fixed points. The translation from this fragment of the modal μ-calculus to simulation logic is straightforward and replaces each fixed point by an equation. As an example, the formula $\nu X.p_1 \wedge (\nu Y.X \wedge [a](p_2 \vee Y))$ is translated into the equivalent formula $X[X = p_1 \wedge Y, Y = X \wedge [a](p_2 \vee Y)]$ of simulation logic. The translation in the other direction is based on Bekič's principle (cf. [34,31]), which expresses a fixed point in a product lattice in terms of a vector of component-wise fixed points.

2.3. Weak simulation and logic

Often, one is only interested in the *observable* behaviour of systems. To achieve this, one can identify a distinguished action $\varepsilon \in A$, called the *silent* action, and define *weak* transitions $s \stackrel{a}{\Rightarrow} t$ in terms of the usual (strong) transitions as follows: $s \stackrel{\varepsilon}{\Rightarrow} t$ whenever $s(\stackrel{\varepsilon}{\rightarrow})^* t$, and $s \stackrel{a}{\Rightarrow} t$ whenever $s \stackrel{\varepsilon}{\Rightarrow} \stackrel{a}{\rightarrow} \stackrel{\varepsilon}{\Rightarrow} t$ for all $a \neq \varepsilon$. Weak simulation $\leqslant_w$ (weak simulation equivalence $\simeq_w$) is then defined as simulation (simulation equivalence) with respect to weak transitions. Similarly, we can interpret the box modality of simulation logic over the weak transitions rather than the strong transitions of models. To distinguish the two interpretations, we shall redefine the notion of satisfaction and write $\mathcal{S} \models_w \phi$ in that case. Thus, $\mathcal{S} \models_w [a]\phi$ holds if and only if all states that can be reached from some entry state of $\mathcal{S}$ by a transition labelled a, preceded and followed by an arbitrary number of ε-steps, satisfy ϕ.

Example 7. Consider again the specification in Fig. 1. Then $(\mathcal{M}, \{s_1\}) \models_w [\varepsilon]p$, but not $(\mathcal{M}, \{s_3\}) \models_w [a]q$, since $s_3 \stackrel{a}{\Rightarrow} s_2$ but s_2 does not satisfy the atomic proposition q.

3. Representation results

This section relates simulation logic to simulation by defining two mappings, χ and θ. The mapping χ translates each *finite* specification into a formula, while θ translates formulae into (finite) specifications. The latter map is first defined on formulae in so-called simulation normal form (SNF), and is then extended to all formulae by showing how any formula can be transformed into an equivalent one in SNF. We show that χ logically characterises simulation and θ behaviourally characterises logical satisfaction. These two maps form a Galois connection between finite specifications ordered by simulation and formulae ordered by logical consequence. Similar results for somewhat different settings appear in [22,23,9]. In this paper, we present a novel procedure to construct maximal models, which is similar to the construction by Boudol and Larsen [22], but handles greatest fixed points. In contrast to constructions for other branching-time logics [6,21], we do not directly build the model, but proceed by a step-wise transformation of the formula into an equivalent one in SNF, which is isomorphic to the desired maximal model. Moreover, unlike in constructions for more expressive logics [25,26], our maximal models are representable as standard transition systems. To the best of our knowledge, this is the first maximal model construction for a fragment of the modal μ-calculus including the full expressive power of greatest fixed points.

3.1. Characteristic formulae

First, we define the mapping from finite specifications to formulae. A finite specification $(\mathcal{M}, E)$ is translated into its *characteristic formula* $\chi(\mathcal{M}, E) = \phi_E[\Sigma_{\mathcal{M}}]$, where $\phi_E = \bigvee_{s \in E} X_s$ and $\Sigma_{\mathcal{M}}$ defines X_s for each $s \in S$ by

$$X_s = \bigwedge_{a \in L} [a] \left(\bigvee_{s \stackrel{a}{\rightarrow} t} X_t \right) \wedge \bigwedge_{p \in \lambda(s)} p \wedge \bigwedge_{q \in A - \lambda(s)} \neg q$$

Recall that $\bigvee \varnothing = \mathsf{ff}$ (false) and $\bigwedge \varnothing = \mathsf{tt}$ (true).

Example 8. Consider the specification $\mathcal{S}$ displayed in Fig, 1. Its characteristic formula is $\chi(\mathcal{S}) = (X_{s_1} \vee X_{s_2})[\Sigma]$, where

$$\Sigma = \begin{bmatrix} X_{s_1} & = & [a]\mathsf{ff} \wedge [\varepsilon]X_{s_2} \wedge p \wedge q \\ X_{s_2} & = & [a](X_{s_1} \vee X_{s_3}) \wedge [\varepsilon]\mathsf{ff} \wedge p \wedge \neg q \\ X_{s_3} & = & [a]X_{s_1} \wedge [\varepsilon]X_{s_2} \wedge \neg p \wedge \neg q \end{bmatrix}.$$

We have a variation of an earlier result by Larsen [23], stating that specification $\mathcal{S}_1$ is simulated by the finite specification $\mathcal{S}_2$ whenever $\mathcal{S}_1$ satisfies the characteristic formula of $\mathcal{S}_2$.

Theorem 9. *Let $\mathcal{S}_1, \mathcal{S}_2$ be specifications and suppose $\mathcal{S}_2$ is finite. Then $\mathcal{S}_1 \leqslant \mathcal{S}_2$ if and only if $\mathcal{S}_1 \models \chi(\mathcal{S}_2)$.*

Note that using infinite equation systems this theorem generalises to finitely branching $\mathcal{S}_2$.

3.2. Maximal models

The next step is to define the inverse mapping. Not all formulae correspond directly to a specification, but those in simulation normal form do.

Definition 10 (*Simulation normal form*)**.** A formula $\phi[\Sigma]$ of simulation logic is in *simulation normal form (SNF)* if ϕ has the form $\bigvee \mathcal{X}$ for some finite set $\mathcal{X} \subseteq \mathsf{bv}(\Sigma)$ and all equations in Σ are in the following *state normal form*

$$X = \bigwedge_{a \in L} [a]\left(\bigvee \mathcal{Y}_{X,a}\right) \wedge \bigwedge_{p \in B_X} p \wedge \bigwedge_{q \in A - B_X} \neg q$$

where each $\mathcal{Y}_{X,a} \subseteq \mathsf{bv}(\Sigma)$ is a finite set of variables and each $B_X \subseteq A$ is a set of atomic propositions.

Notice that any characteristic formula $\chi(\mathcal{S})$ is in SNF. From a formula $(\bigvee \mathcal{X})[\Sigma]$ in SNF we derive the specification $\theta((\bigvee \mathcal{X})[\Sigma]) = ((S, L, \rightarrow, A, \lambda), E)$ where $S = \mathsf{bv}(\Sigma)$, $E = \mathcal{X}$ and, for each $X \in \mathsf{bv}(\Sigma)$, the equation for X induces the transitions $\{X \xrightarrow{a} Y \mid Y \in \mathcal{Y}_{X,a}\}$ and the valuation $\lambda(X) = B_X$.

Lemma 11. *χ and θ are each others inverse up to equivalence, that is,*

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

Here, isomorphism means a label-and-valuation-preserving bijection between the respective states and transitions.

For ϕ in SNF, the specification $\theta(\phi)$ is a *maximal model* of ϕ with respect to the simulation preorder, in the sense that it simulates exactly those specifications that satisfy formula ϕ.

Theorem 12. *For ϕ in SNF, we have $\mathcal{S} \leqslant \theta(\phi)$ if and only if $\mathcal{S} \models \phi$.*

Proof. Follows from Theorem 9 by Lemma 11(2). □

3.3. Transformation to SNF

We now present a step-wise transformation of any simulation logic formula into a logically equivalent formula in SNF. Before describing the transformation in detail, we introduce some auxiliary notions. First, we use a slightly non-standard variant of disjunctive normal form: we say that a formula ϕ of basic simulation logic is in *disjunctive normal form* (*DNF*) if it is a disjunction of conjunctions of box formulae and literals, i.e., it has the shape $\phi = \bigvee_i (\bigwedge_j [a_{ij}]\psi_{ij} \wedge \bigwedge \mathcal{L}_i)$ where $\mathcal{L}_i$ are sets of literals and ψ_{ij} arbitrary formulae in basic simulation logic. Furthermore, the *conjunctive decomposition* $c(\psi)$ of a formula ψ into its conjuncts is given by $c(\psi) = \{\psi_1, \ldots, \psi_m\}$ such that no ψ_i is a conjunction and $\psi = \bigwedge_i \psi_i$ (modulo associativity and commutativity). Note that $c(\mathsf{tt}) = \varnothing$. The elements of $c(\psi)$ are called *conjunctive components* of ψ.

We call an occurrence of a subformula *top-level* if it is not under the scope of a box operator. We say that Y is *unguarded* in ϕ_X, written $X \triangleright Y$, if there is a top-level occurrence of Y in ϕ_X. A modal equation system Σ (or formula $\phi[\Sigma]$) is *weakly guarded* if the relation $\triangleright$ is acyclic, and *strongly guarded* if $\triangleright$ is empty.

Example 13. Consider the modal equation system

$$\Sigma = \begin{bmatrix} X & = & [a]X \vee (q \wedge Y) \\ Y & = & [b](X \wedge [a]Y) \wedge p \end{bmatrix}$$

Variable X is guarded in ϕ_X (the only occurrence of X is under the scope of a box operator), but Y is not (it occurs on the top-level). Both X and Y are guarded in ϕ_Y. Hence, $\triangleright = \{(X, Y)\}$ being acyclic but not empty, Σ is weakly guarded but not strongly guarded.

Any weakly guarded formula can be transformed into a strongly guarded one by repeatedly rewriting each unguarded occurrence of a variable by its defining equation. Moreover, using a result of Walukiewicz [35] we can also show that any formula of simulation logic can be transformed into an equivalent weakly guarded one (and thus into a strongly guarded one).

After these auxiliary definitions, we are ready to present the transformation. It consists of three phases:

Phase I transforms each equation into a disjunction of formulae in state normal form, where only single variables appear under modalities,

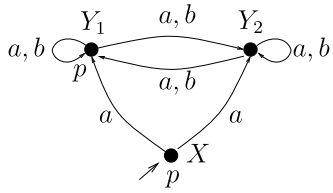

Fig. 2. Maximal model for $\phi = [b]\,\mathsf{ff} \wedge p$.

Phase II splits top-level disjunctions in each equation into a set of new equations, one for each disjunct, yielding an equation system in state normal form, and

Phase III is an optimisation phase removing unreachable and redundant equations.

The transformation into SNF uses a partial function h that keeps track how sets of formulae are mapped to variables. This map avoids the repeated introduction of new equations for the same formula, which is essential for the termination of the transformation. If h maps a set of formulae Ψ to variable X, this means that an equation $X = \bigwedge \Psi$ (such that $c(\bigwedge \Psi) = \Psi$) has been introduced earlier and that variable X should be used instead of introducing any further equation for $\bigwedge \Psi$.

Before going into the details, let us illustrate the basic ideas on a simple example. A more elaborate transformation example appears in Section 8.3.1.

Example 14. Let $\phi = [b]\,\mathsf{ff} \wedge p$ be interpreted as a formula over $L = \{a, b\}$ and $A = \{p\}$. This formula holds for specifications, where each initial state satisfies p and has no outgoing b transition. We first translate ϕ to $(\bigvee \mathcal{X}_0)[\Sigma_0]$ with $\mathcal{X}_0 = \{X\}$ and $\Sigma_0 = \{X = [b]\,\mathsf{ff} \wedge p\}$. In the following, the numbers in parentheses refer to the transformation steps detailed below.

The equation for X is already strongly guarded (I.1) and in DNF (I.2). Next, we add the missing box $[a]$ using the equivalence $\mathsf{tt} \equiv [a]\,\mathsf{tt}$ (I.3), yielding $X = [a]\,\mathsf{tt} \wedge [b]\,\mathsf{ff} \wedge p$. In the next step (I.4), we introduce new variables for the formulae under the boxes: $Y = \mathsf{tt}$ and $Z = \mathsf{ff}$. This is recorded in h with two new entries: $(\varnothing, Y)$ (since $\mathsf{tt} = \bigwedge \varnothing$) and $(\{\mathsf{ff}\}, Z)$. The equation for X becomes

$$X = [a]\,Y \wedge [b]\,Z \wedge p$$

which is already in state normal form. We proceed with $Y = \mathsf{tt}$. Again, the first step with an effect adds the missing boxes (I.3), producing $Y = [a]\,\mathsf{tt} \wedge [b]\,\mathsf{tt}$. Next, since $c(\mathsf{tt}) = \varnothing$ and $h(\varnothing) = Y$, we know that Y stands for tt, so we replace the subformulae tt under the boxes by Y, yielding $Y = [a]\,Y \wedge [b]\,Y$. To get a disjunction of state normal forms, we add the missing literals in positive and negative form, yielding

$$Y = ([a]\,Y \wedge [b]\,Y \wedge p) \vee ([a]\,Y \wedge [b]\,Y \wedge \neg p).$$

The third equation $Z = \mathsf{ff}\ (= \bigvee \varnothing)$ is already a (trivial) disjunction of state normal forms. Note that $\mathcal{X}$ remains unchanged in Phase I. Thus, at the end of Phase I we have the following equation system.

$$\Sigma = \begin{bmatrix} X & = & [a]\,Y \wedge [b]\,Z \wedge p \\ Y & = & ([a]\,Y \wedge [b]\,Y \wedge p) \vee ([a]\,Y \wedge [b]\,Y \wedge \neg p) \\ Z & = & \mathsf{ff} \end{bmatrix}$$

Next, Phase II splits each top-level disjunction into a set of new equations and substitutes the disjunction of new variables for the original variable. Concretely, all occurrences of Y are replaced by $Y_1 \vee Y_2$ and $Z = \mathsf{ff}\ (= \bigvee \varnothing)$ is substituted back into ϕ_X, yielding

$$\Sigma = \begin{bmatrix} X & = & [a]\,(Y_1 \vee Y_2) \wedge [b]\,\mathsf{ff} \wedge p \\ Y_1 & = & [a]\,(Y_1 \vee Y_2) \wedge [b]\,(Y_1 \vee Y_2) \wedge p \\ Y_2 & = & [a]\,(Y_1 \vee Y_2) \wedge [b]\,(Y_1 \vee Y_2) \wedge \neg p \end{bmatrix}$$

Since X is not split into several equations, $\mathcal{X} = \{X\}$ remains unchanged. Phase III is the identity transformation in this example as there are no unreachable or duplicate equations. Thus, the final result is $X[\Sigma]$, which is in simulation normal form. The derived maximal model $\theta(X[\Sigma])$ is displayed in Fig. 2. Indeed, it simulates exactly all those specifications where each initial state satisfies p and has no outgoing b transition.

We now describe the actual transformation in detail. We assume without loss of generality that the initial formula has the shape $X_0[\Sigma_0]$, where Σ_0 is weakly guarded (since any formula can be transformed into a weakly guarded one). We initialise $\mathcal{X} = \{X_0\}$, $\Sigma = \Sigma_0$ and $h = \varnothing$.

Phase I (disjunction of state normal forms)

This phase transforms each equation into a disjunction of formulae in state normal form. Its steps are applied once to each equation including the new ones introduced in step I.4 below.

(1) (Strong guardedness) Make equation strongly guarded by repeated rewriting of unguarded occurrences of variables using the original system Σ_0.
(2) (DNF) Put equation into disjunctive normal form and remove inconsistent disjuncts (those where ff or both p and $\neg p$ appear).
(3) (Box grouping and completion) Group boxes together using $[a]\,\phi_1 \wedge [a]\,\phi_2 \equiv [a]\,(\phi_1 \wedge \phi_2)$ and add missing boxes to each disjunct using $\mathsf{tt} \equiv [a]\,\mathsf{tt}$ such that there is a box formula for each $a \in L$. The resulting equation shape is

$$X = \bigvee_i \left(\bigwedge_{a \in L} [a]\,\psi_{ia} \wedge \bigwedge \mathcal{L}_i \right)$$

(4) (Modal depth reduction) Apply the following to each top-level box subformula $[a]\,\psi_{ia}$ where ψ_{ia} is not a variable. If $(c(\psi_{ia}), Y) \in h$ for some variable Y then replace $[a]\,\psi_{ia}$ by $[a]\,Y$; otherwise, choose a fresh variable $Z \notin \mathsf{bv}(\Sigma)$, add the new equation $Z = \psi_{ia}$ to Σ, replace $[a]\,\psi_{ia}$ by $[a]\,Z$ and extend h to $h \cup \{(c(\psi_{ia}), Z)\}$. The equation shape is then

$$X = \bigvee_i \left(\bigwedge_{a \in L} [a]\,Z_{ia} \wedge \bigwedge \mathcal{L}_i \right)$$

(5) (Literal completion) Replace equation $X = \phi$ by $X = \phi \wedge \bigwedge_{p \in A}(p \vee \neg p)$, then repeat step (2) to put equation back into DNF. The equation shape is (for $B_i \subseteq A$)

$$X = \bigvee_i \left(\bigwedge_{a \in L} [a]\,Z_{ia} \wedge \bigwedge_{p \in B_i} p \wedge \bigwedge_{q \in A - B_i} \neg q \right)$$

Note that step (I.4) might introduce unguarded occurrences of variables in the newly added equations. Thus, the rewriting step (I.1) is needed to bring these equations into strongly guarded form. For the termination of Phase I, it is crucial to use the original equation system Σ_0 and not the current Σ in this step, because this limits the set of subformulae introduced by the rewriting to those already occurring in Σ_0. This in turn guarantees that subsequent modal depth reductions in step (I.4) eventually find already existing variables for the subformulae under the box operator.

Phase II (push disjunctions inside)

This phase eliminates the top-level disjunctions by introducing a new equation for each disjunct, thus pushing these disjunctions under box modalities. It is applied once to each equation in Σ.

(1) Remove an equation of shape $X = \bigvee_{i=1}^{n} \phi_i$ with $n \neq 1$ from Σ; note that this includes the case $X = \mathsf{ff}$ (for $n = 0$).
(2) Add a new equation $X_i = \phi_i$ for each non-variable disjunct ϕ_i and substitute $\bigvee_{i=1}^{n} X_i$ for X in all equations of Σ (where X_i is either identical to ϕ_i or X_i is the fresh variable introduced for ϕ_i).
(3) If $X \in \mathcal{X}$ then replace $\mathcal{X}$ by $(\mathcal{X} - \{X\}) \cup \{X_1, \ldots, X_n\}$.

The resulting equation is in state normal form.

Phase III (optimisation)

This optimisation phase iteratively removes unreachable and duplicate equations.

(1) Remove equations $Z = \psi$ from Σ in case Z can not be reached from any variable in $\mathcal{X}$ via variable dependencies (X depends on Y if Y occurs in ϕ_X).
(2) If there are equations $Z_1 = \psi_1$ and $Z_2 = \psi_2$ in Σ such that $\psi_1[Z_1/Z_2] = \psi_2[Z_1/Z_2]$, then remove $Z_2 = \psi_2$ from Σ and substitute Z_1 for Z_2 in the remaining equations as well as in $\mathcal{X}$.

Theorem 15. *The algorithm above terminates and transforms any formula ϕ of simulation logic into an equivalent formula* $\mathsf{snf}(\phi)$ *in simulation normal form.*

Proof. (Sketch; full proof in [36]) Let $\mathcal{X}_i$, Σ_i and h_i denote the values of $\mathcal{X}$, Σ and h after i transformation steps. We concentrate in this sketch on Phase I, which preserves the following two invariants:

J1. for all $Y \in \mathsf{bv}(\Sigma_0)$ we have $Y \in \mathsf{bv}(\Sigma_i)$ and $Y[\Sigma_i] \equiv Y[\Sigma_0]$, and

J2. if $(\Psi, Z) \in h_i$ then $\Psi \subseteq \Psi_0$, where Ψ_0 is defined as the set of conjunctive components of subformulae appearing under some box modality in Σ_0, that is, $\Psi_0 = \bigcup\{c(\psi) \mid \exists a.\, [a]\psi$ is a subformula of $\Sigma_0\}$.

Preservation of the semantics by the transformation steps follows from J1 and the fact that $\mathcal{X}$ is constant in Phase I. To see that Phase I terminates, note first that step I.1 terminates, because Σ_0 is weakly guarded (by assumption) and all steps preserve weak guardedness. Overall non-termination of Phase I due to the introduction of equations in step I.4 is ruled out by J2: since Ψ_0 is finite, the map h eventually fills up and thus Phase I terminates. □

We extend the mapping θ to all formulae of simulation logic by defining $\theta(\phi) = \theta(\mathsf{snf}(\phi))$. Since snf preserves the semantics, Theorem 12 can be extended to all formulae, showing that $\theta(\phi)$ is the maximal model of ϕ with respect to the simulation preorder.

Theorem 16. *$\mathcal{S} \leqslant \theta(\phi)$ if and only if $\mathcal{S} \models \phi$.*

We conclude with two important consequences of Theorems 9 and 16. The first one is that simulation preserves logical properties.

Corollary 17. *$\mathcal{S}_1 \leqslant \mathcal{S}_2$ and $\mathcal{S}_2 \models \phi$ imply $\mathcal{S}_1 \models \phi$.*

The second corollary expresses that the maps χ and θ form a Galois connection between the preorder $(\mathsf{S}, \leqslant)$ of (isomorphism classes of) *finite* specifications ordered by simulation and be the preorder $(\mathsf{L}, \sqsubseteq)$ of formulae of simulation logic ordered by logical consequence.

Corollary 18. *χ and θ are monotone and, for finite specifications $\mathcal{S}$, $\mathcal{S} \leqslant \theta(\phi)$ if and only if $\chi(\mathcal{S}) \sqsubseteq \phi$.*

3.4. Representation results for weak simulation

A natural question is whether the results of the previous subsection can be used to relate weak simulation and simulation logic in the same way as simulation and simulation logic are related by the transformation θ (and its adjoint map χ). Note that applying θ on a formula of simulation logic interpreted over weak transitions would only give us a model in terms of weak transitions, without the underlying strong transitions. However, there is a standard translation of formulae interpreted over weak transitions into equivalent formulae interpreted over strong transitions [32]. This translation, let us denote it by δ, is easily adapted to our setting. It has the property that $\mathcal{S} \models_w \phi$ exactly when $\mathcal{S} \models \delta(\phi)$. We show that $\theta \circ \delta$ provides the desired transformation relating weak simulation and simulation logic.

To this end, we first introduce the notion of saturated model, i.e., a model in which $s \xrightarrow{a} t$ whenever $s \stackrel{a}{\Rightarrow} t$. We show that for all formulae ϕ, $\theta(\delta(\phi))$ is simulation equivalent to its saturation, and therefore it is sufficient for a model to be weakly simulated by $\theta(\delta(\phi))$ in order to satisfy ϕ when interpreted over weak transitions.

Definition 19 (*Saturation*)**.** Let $\mathcal{M} = (S, L, \rightarrow, A, \lambda)$ be a model. The *saturation* of $\mathcal{M}$ is the model $sat(\mathcal{M}) = (S, L, \rightarrow_s A, \lambda)$ in which $s \xrightarrow{a}_s t$ exactly when $s \stackrel{a}{\Rightarrow} t$ for all a. The saturation of a specification $(\mathcal{M}, E)$ is the specification $sat(\mathcal{M}, E) = (sat(\mathcal{M}), E)$.

Thus, $sat(\mathcal{M})$ is the least saturated model with respect to the subset ordering on the powerset of $S \times L \times S$, containing $\mathcal{M}$. For instance, in the model given in Fig. 1 above, we have to add the transition $s_i \xrightarrow{a} s_j$ for all i and j and ε-self-loops to saturate the model. We have $s \stackrel{a}{\Rightarrow}_s t$ in $sat(\mathcal{S})$ whenever $s \xrightarrow{a}_s t$ in $sat(\mathcal{S})$ whenever $s \stackrel{a}{\Rightarrow} t$ in $\mathcal{S}$. As consequences, we have the following properties of weak simulation and simulation logic.

Proposition 20. *We have*
(i) $\mathcal{S}_1 \leqslant_w \mathcal{S}_2$ iff $\mathcal{S}_1 \leqslant sat(\mathcal{S}_2)$, and
(ii) $\mathcal{S} \models_w \phi$ iff $sat(\mathcal{S}) \models_w \phi$ iff $\mathcal{S} \models \delta(\phi)$.

Lemma 21. *$sat(\theta(\delta(\phi))) \simeq \theta(\delta(\phi))$.*

Proof. Clearly, $\theta(\delta(\phi)) \leqslant sat(\theta(\delta(\phi)))$ holds; it remains to show the other direction. From reflexivity of $\leqslant$ and Theorem 16 we know that $\theta(\delta(\phi)) \models \delta(\phi)$. Then, by Proposition 20(ii), $sat(\theta(\delta(\phi))) \models \delta(\phi)$, and again by Theorem 16, $sat(\theta(\delta(\phi))) \leqslant \theta(\delta(\phi))$. □

These results allow the following characterisation of simulation logic, in the style of Theorem 16.

Theorem 22. *$\mathcal{S} \leqslant_w \theta(\delta(\phi))$ if and only if $\mathcal{S} \models_w \phi$.*

Proof. By Proposition 20(i) and Lemma 21 the following statements are equivalent: (a) $\mathcal{S} \leqslant_w \theta(\delta(\phi))$, (b) $\mathcal{S} \leqslant sat(\theta(\delta(\phi)))$, and (c) $\mathcal{S} \leqslant \theta(\delta(\phi))$. Theorem 16 together with Proposition 20(ii) then establish the result. □

Corollary 23. *$\mathcal{S}_1 \leqslant_w \mathcal{S}_2$ and $\mathcal{S}_2 \models_w \phi$ imply $\mathcal{S}_1 \models_w \phi$.*

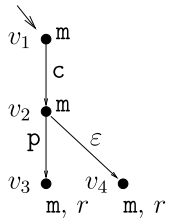

Fig. 3. A method graph.

4. Program model

This section uses the notions developed above to formally define applet structure and behaviour, structural and behavioural simulation logic, and maximal applets. The next section then shows how these support compositional verification of control-flow-based safety properties of applets.

4.1. Applet structure

We model the control structure of an applet as a collection of method specifications. We first define the notion of applet interface as the sets of methods which are provided and called by an applet. We shall need this notion for constructing maximal applets. Let $\mathcal{M}eth$ be an infinite set of method names (not containing the special symbols r and ε).

Definition 24 (*Applet interface*)**.** An *applet interface* is a pair $I = (I^+, I^-)$, where $I^+, I^- \subseteq \mathcal{M}eth$ are finite sets of names of *provided* and *required* methods, respectively. We say I is *closed* if $I^- \subseteq I^+$. The *composition* of two interfaces $I_1 = (I_1^+, I_1^-)$ and $I_2 = (I_2^+, I_2^-)$ is defined by $I_1 \cup I_2 = (I_1^+ \cup I_2^+, I_1^- \cup I_2^-)$.

Next, we define method specifications, which are the basic building blocks of applets. Each method is described by its control-flow graph and a set of entry points.

Definition 25 (*Method specification*)**.** A *method graph* for $m \in \mathcal{M}eth$ over a set M of method names is a finite model $\mathcal{M}_m = (V_m, L_m, \rightarrow_m, A_m, \lambda_m)$, where V_m is the set of control nodes of m, $L_m = M \cup \{\varepsilon\}$, $A_m = \{m, r\}$, $m \in \lambda_m(v)$ for all $v \in V_m$, i.e., each node is tagged with the method name. A *method specification* for $m \in \mathcal{M}eth$ over M is a specification $(\mathcal{M}_m, E_m)$ such that $\mathcal{M}_m$ is a method graph for m over M.

The nodes labelled with the distinguished atomic proposition r are the *return points* of m.

Example 26. Fig. 3 shows the method graph for the following Java-like method m:

```
void m() {if c() {p()} else x = 3}
```

An applet is a collection of method specifications.

Definition 27 (*Applet*)**.** *Applets $\mathcal{A}$ with interface I*, written $\mathcal{A} : I$, are inductively defined by

- $\mathbf{0}_M : (\varnothing, M)$, where $\mathbf{0}_M$ is the *empty applet* over M defined by $\mathbf{0}_M = ((\varnothing, M \cup \{\varepsilon\}, \varnothing, \{r\}, \varnothing), \varnothing)$,
- $(\mathcal{M}_m, E_m) : (\{m\}, M)$ if $(\mathcal{M}_m, E_m)$ is a method specification for m over M,
- $\mathcal{A}_1 \uplus \mathcal{A}_2 : I_1 \cup I_2$ if $\mathcal{A}_1 : I_1$ and $\mathcal{A}_2 : I_2$.

An applet $\mathcal{A} : I$ is *closed* if its interface I is closed.

This definition requires that each provided method $m \in I^+$ of an applet $\mathcal{A} : I$ has to be implemented in a method graph for m. The interface of an applet can be derived from its implementation: a straightforward induction shows that if $\mathcal{A}$ is an applet built from a model over L and A then its interface is $(A - \{r\}, L - \{\varepsilon\})$. We write $\mathcal{S} : I$ for an arbitrary specification $\mathcal{S}$ to mean that $\mathcal{S}$ is (isomorphic to) an applet with interface I. Note that, up to isomorphism, applet composition $\uplus$ is associative and commutative with neutral element $\mathbf{0}_\varnothing$.

We have developed a tool to extract applet graphs from Java Card byte code. The tool is based on the SOOT framework (see Section 7).

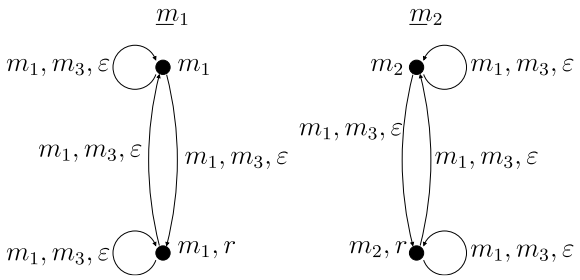

Fig. 4. Maximal applet for interface $I = (\{m_1, m_2\}, \{m_1, m_3\})$ and $\phi = \mathrm{tt}$.

4.1.1. Structural simulation and logic

Structural simulation on applets coincides with simulation on the specifications defining the applets. For convenience we write $\mathcal{A}_1 \leqslant_s \mathcal{A}_2$ instead of $\mathcal{A}_1 \leqslant \mathcal{A}_2$ to denote *structural simulation*. Since applet composition corresponds to disjoint union, structural simulation is preserved by applet composition (cf. Theorem 4).

Corollary 28. *If $\mathcal{A}_1 \leqslant_s \mathcal{B}_1$ and $\mathcal{A}_2 \leqslant_s \mathcal{B}_2$ then $\mathcal{A}_1 \uplus \mathcal{A}_2 \leqslant_s \mathcal{B}_1 \uplus \mathcal{B}_2$.*

We also instantiate (weak) simulation logic to this level. For an applet $\mathcal{A} : I$ and a formula ϕ of simulation logic over $L = I^- \cup \{\varepsilon\}$ and $A = I^+ \cup \{r\}$ we write for clarity $\mathcal{A} \models_s \phi$ instead of $\mathcal{A} \models \phi$ and $\mathcal{A} \models_{s,w} \phi$ instead of $\mathcal{A} \models_w \phi$.

4.2. Maximal applet structures

In general, the maximal model of a given formula in structural simulation logic is not a legal applet structure. What we are interested in, then, is computing a *maximal applet* for the formula, i.e., an applet structure which satisfies the formula and which structurally simulates all other applets satisfying the formula. This problem, however, can only be solved for a fixed applet interface: one can axiomatise applet structures within structural simulation logic for a given interface. This allows the maximal model construction presented above to be used for computing a maximal applet for a given formula in structural simulation logic.

Definition 29 (*Interface formula*)**.** Let $I = (I^+, I^-)$ be an applet interface. Define $\phi_I[\Sigma_I]$, the *interface formula for I*, by

$$\begin{array}{lcl} \phi_I & = & \bigvee_{m \in I^+} X_m \\ \Sigma_I & = & \{X_m = [I^-, \varepsilon] X_m \wedge p_m \mid m \in I^+\} \\ p_m & = & m \wedge \bigwedge \{\neg m' \mid m' \in I^+, m' \neq m\} \end{array}$$

The formula $\phi_I[\Sigma_I]$ axiomatises the basic structure of an applet with interface I, namely, each initial node belongs to a unique method m and no transition leaves m. Note that Σ_I is *not* in SNF (proposition r is missing).

The maximal applet with respect to a formula ϕ and interface I is defined as the maximal model of ϕ conjoined with the interface formula for I.

Definition 30 (*Maximal applet*)**.** The *maximal applet* with respect to interface I and formula $\phi[\Sigma]$ is defined as $\theta_I(\phi[\Sigma]) = \theta(\phi \wedge \phi_I[\Sigma, \Sigma_I])$ (where it is assumed without loss of generality that the bound variables of Σ and Σ_I are disjoint).

Example 31. The interface formula for interface $I = (\{m_1, m_2\}, \{m_1, m_3\})$ is given by the formula $\phi_I[\Sigma_I]$, where $\phi_I = X_{m_1} \vee X_{m_2}$ and

$$\Sigma_I = \begin{bmatrix} X_{m_1} = [m_1, m_3, \varepsilon] X_{m_1} \wedge m_1 \wedge \neg m_2 \\ X_{m_2} = [m_1, m_3, \varepsilon] X_{m_2} \wedge m_2 \wedge \neg m_1 \end{bmatrix}$$

The maximal applet for interface I (and formula $\phi = \mathrm{tt}$) is shown in Figure 4.

The following result records the main properties of interface formulae and maximal applets.

Theorem 32. *Let I be an applet interface. For any specification $\mathcal{S} = (\mathcal{M}, E)$ over labels $L = I^- \cup \{\varepsilon\}$ and atomic propositions $A = I^+ \cup \{r\}$ we have (where $\mathcal{R}$ denotes the reachable part of a specification, as defined on page 844)*

(1) $\mathcal{S} \models_s \phi_I$ if and only if $\mathcal{R}(\mathcal{S}) : I$, and

(2) $\mathcal{S} \leqslant_s \theta_I(\phi)$ if and only if $\mathcal{S} \models_s \phi$ and $\mathcal{R}(\mathcal{S}) : I$.

Proof. (1) (Sketch) "⇒" By an induction on the size of I^+. The restriction to the reachable part of $\mathcal{S}$ is required, because the formula ϕ_I does not constrain the unreachable parts of $\mathcal{S}$. "⇐" By inspection of the definition of applets. (2) Using the definition of $\theta_I(\phi)$ and Theorem 16 we know that $\mathcal{S} \leqslant_s \theta_I(\phi)$ is equivalent to $\mathcal{S} \models_s \phi$ and $\mathcal{S} \models_s \phi_I$. The result then follows from (1). □

Point (1) of the theorem essentially expresses that the formula ϕ_I characterises those specifications that are applets with interface I, while point (2) extends Theorem 16 from specifications to applets. As a consequence of (2) we have $\theta_I(\phi) \models \phi_I$ and $\theta_I(\phi) : I$, since all nodes of $\theta_I(\phi)$ are reachable by construction.

4.3. Applet behaviour

Next, we change our focus to the behavioural level, where we first define the operational semantics of a closed applet. Since our compositional verification method is based on structural assumptions, there is no need to compose applets on the behavioural level, so an operational semantics of *closed* applets is sufficient. This is in contrast with previous work on semi-automatic compositional applet verification [7] where the use of behavioural assumptions required a more involved open semantics of applets.

Applet behaviour can be described in terms of Pushdown Automata. We also present an equivalent formulation of applet behaviour, defining it directly in terms of a model. Applet behaviour is closely connected with applet structure, in the sense that simulation of applet structure immediately carries over to simulation of applet behaviours. This will be exploited in the next section, when presenting the compositional verification principle.

4.3.1. Applet behaviour as Pushdown Automaton

Pushdown Automata provide a natural execution model for programs with recursion. They form a well-studied class of infinite state systems for which many important problems like bisimulation equivalence and model checking are decidable (see e.g., [10,5] for analysis techniques and [3,2] for applications). Applet behaviour can be described directly in terms of Pushdown Automata.

Definition 33 (*PDA*)**.** A *non-deterministic Pushdown Automaton* is a tuple $\mathcal{P} = (Q, \Sigma, \Gamma, \Delta, Q', \bot)$ where Q is a set of control states, Σ a finite input alphabet, Γ a finite stack alphabet, $Q' \subseteq Q$ are the start states, $\bot \in \Gamma$ is the initial stack symbol, and $\Delta \subseteq (Q \times \Gamma) \times \Sigma \times (Q \times \Gamma^*)$ a set of labelled productions (or rewrite rules) of the shape $(q_1, A) \overset{a}{\hookrightarrow} (q_2, \gamma)$.

A *configuration* of a PDA is a pair $(q, \gamma) \in Q \times \Gamma^*$. The set of configurations $Q' \times \{\bot\}$ are called *initial configurations*. The set of productions induces a labelled transition relation on configurations as the least relation which contains the initial configurations and is closed under the *prefix rewrite rule*: $(q_1, A \cdot \gamma') \overset{a}{\to} (q_2, \gamma \cdot \gamma')$ whenever $(q_1, A) \overset{a}{\hookrightarrow} (q_2, \gamma) \in \Delta$.

Applet behaviour is induced from the *applet PDA* through the prefix rewrite rule. The connection between applet structure and applet PDA is established through the following definition.

Definition 34 (*Applet PDA*)**.** Let $\mathcal{A} = (\mathcal{M}, E) : (I^+, I^-)$ be a closed applet such that $\mathcal{M} = (V, L, \to, A, \lambda)$. Then $\mathcal{P}_\mathcal{A} = (V, L_b, V \cup \{\bot\}, \Delta, E, \bot)$ is the PDA induced by $\mathcal{A}$ where

$$
\begin{array}{lcl}
L_b & = & \{m_1\ l\ m_2 \mid l \in \{\mathsf{call}, \mathsf{ret}\},\ m_1, m_2 \in I^+\} \cup \{\varepsilon\} \\
\Delta & = & \{(v, v_\bot) \overset{\varepsilon}{\hookrightarrow} (v', v_\bot) \mid v \models \neg r \wedge v \to_m v'\} \\
& \cup & \{(v_1, v_\bot) \overset{m_1\,\mathsf{call}\,m_2}{\hookrightarrow} (v_2, v_1' \cdot v_\bot) \mid v_1 \models \neg r \wedge v_1 \xrightarrow[m_1]{m_2} v_1' \\
& & \qquad\qquad \wedge\, v_2 \models m_2 \wedge v_2 \in E\} \\
& \cup & \{(v_2, v_1) \overset{m_2\,\mathsf{ret}\,m_1}{\hookrightarrow} (v_1, \varepsilon) \mid v_2 \models r \wedge v_2 \models m_2 \wedge v_1 \models m_1\}
\end{array}
$$

where $v_\bot$ ranges over $V \cup \{\bot\}$.

Note that the valuation λ also applies to PDA control states and is lifted to configurations by defining $\hat{\lambda}((v, v_\bot)) = \lambda(v)$.

4.3.2. Applet behaviour by transition rules

An alternative approach is to describe applet behaviour explicitly as a specification, by defining appropriate transition rules.

Definition 35 (*Behaviour*)**.** Let $\mathcal{A} = (\mathcal{M}, E) : (I^+, I^-)$ be a closed applet such that $\mathcal{M} = (V, L, \to, A, \lambda)$. The *behaviour* of $\mathcal{A}$ is described by the specification $b(\mathcal{A}) = (\mathcal{M}_b, E_b)$, where $\mathcal{M}_b = (S_b, L_b, \to_b, A_b, \lambda_b)$ is defined by $S_b = V \times V^*$, that is, states are

Table 1
Applet transition rules

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

pairs of control points and stacks, $L_b = \{m_1\ l\ m_2 \mid l \in \{\mathsf{call}, \mathsf{ret}\},\ m_1, m_2 \in I^+\} \cup \{\varepsilon\}$, $\rightarrow_b$ is defined by the transition rules of Table 1, $A_b = A$ and $\lambda_b((v,\sigma)) = \lambda(v)$. The set of initial states E_b is defined by $E_b = E \times \{\varepsilon\}$.

A simple inspection of the rules in Table 1 and Definition 34 shows that the behaviour induced by the applet PDA through the prefix rewrite rule is isomorphic to the explicitly described applet behaviour.

4.3.3. Behavioural simulation and logic

Applet $\mathcal{A}_1$ *behaviourally simulates* applet $\mathcal{A}_2$, written $\mathcal{A}_1 \leqslant_b \mathcal{A}_2$, if $b(\mathcal{A}_1) \leqslant b(\mathcal{A}_2)$. Similarly, we instantiate simulation logic on the behavioural level. Behavioural properties are more abstract than structural ones as they do not refer to the program control structure. We define *behavioural satisfaction* $\mathcal{A} \models_b \psi$ as $b(\mathcal{A}) \models \psi$ for applets $\mathcal{A} : I$ and ψ a formula of simulation logic over L_b and A_b. Similarly, *weak behavioural satisfaction* $\mathcal{A} \models_{b,w} \psi$ is defined as $b(\mathcal{A}) \models_w \psi$. Since applet behaviour coincides with behaviour of a Pushdown Automaton, verifying goals of the shape $\mathcal{A} \models_b \psi$ (or $\mathcal{A} \models_{b,w} \psi$) can be reduced to PDA model checking, for which standard algorithms exist.

4.3.4. Simulation correspondence

The notions of applet structure and behaviour have been defined so as to ensure that any two applets related by structural simulation are also related by behavioural simulation. In general, the inverse does not hold, because due to recursion, method graphs can contain nodes that are never reachable at the behavioural level.

Theorem 36 *(Simulation correspondence). If* $\mathcal{A}_1 \leqslant_s \mathcal{A}_2$ *then* $\mathcal{A}_1 \leqslant_b \mathcal{A}_2$.

Proof. Let R be a structural simulation between $\mathcal{A}_1$ and $\mathcal{A}_2$. We lift R from the structural level to R_b on the behavioural level by defining $((v,\sigma),(v',\sigma')) \in R_b$ if and only if $(v,v') \in R$, $|\sigma| = |\sigma'|$ and $(\sigma(i),\sigma'(i)) \in R$ for all $0 \leqslant i < |\sigma|$. It is easy to check that R_b is a behavioural simulation between $\mathcal{A}_1$ and $\mathcal{A}_2$. □

As a consequence, in the set of applets satisfying a given structural formula $\phi[\Sigma]$, the maximal applet for this formula (with respect to structural simulation) $\theta_I(\phi[\Sigma])$ is also maximal with respect to behavioural simulation.

4.4. Behavioural maximal applets

Defining the maximal applet behaviour for a given behavioural formula is more problematic. As in the structural case, in general, the maximal model of a formula in behavioural simulation logic is not a legal applet behaviour. Unlike the structural case, however, one cannot axiomatise applet behaviour within behavioural simulation logic (in order to use the maximal model construction for generating maximal applet behaviours), since simulation logic is only able of capturing regular properties and not the context-free properties exhibited by Pushdown Automata.

Furthermore, a maximal applet behaviour would in general be infinite-state; therefore, a maximal behaviour construction has to return a finite representation of this behaviour. The obvious (but not only) choice for such a representation would be an applet structure. Given a formula in behavioural simulation logic, the problem then reduces to finding an applet which satisfies the formula *and* which behaviourally simulates all other applets satisfying the formula. However, in general such a maximal applet is not unique.

Example 37. Consider the behavioural formula $[m_1\ \mathsf{call}\ m_2]\,r$ over an interface $I = (\{m_1, m_2\}, \{m_1, m_2\})$. The formula gives rise to two maximal applets:

(1) the maximal applet for I, but without edges labelled m_2 whose source is a non-return entry node of m_1 *(*representable as $\theta_I(\neg m_1 \vee r \vee [m_2]\,\mathsf{ff})$*)*, i.e., the applet where m_1 can never call m_2 immediately; and
(2) the maximal applet for I, but where every entry point of m_2 is valuated r *(*representable as $\theta_I(\neg m_2 \vee r)$*)*, i.e., the applet where m_2 always returns immediately.

Every applet satisfying the formula is behaviourally simulated by one of these two applets; however, neither of the two applets simulates the other.

We are currently investigating under what conditions and how such a collection of maximal applets can be characterised exactly, by means of a translation from behavioural properties into collections of structural properties. Preliminary results are presented by Gurov and Huisman in [11].

5. Compositional verification

The results of the preceding sections form the basis for compositional verification of applets using maximal models.

5.1. Structural properties

In the realm of structural properties, i.e., when global guarantees and local assumptions are all given as structural formulae, we obtain a compositional verification principle of the desired form, embodied by the following rule:

$$(\mathsf{struct-comp})\ \frac{\mathcal{A} \models_s \phi \qquad \theta_I(\phi) \uplus \mathcal{B} \models_s \psi}{\mathcal{A} \uplus \mathcal{B} \models_s \psi}\ \mathcal{A} : I$$

This principle states that in order to show that a composed applet $\mathcal{A} \uplus \mathcal{B}$ has a structural property ψ, it is sufficient to find a structural property ϕ which is satisfied by $\mathcal{A}$ and for which $\theta_I(\phi) \uplus \mathcal{B} \models_s \psi$. The rule is sound and complete. The proof of this rule follows closely the (slightly more involved) proof for rule (compos) presented below (Theorem 39), and is therefore omitted. Verifying the premises is achieved by standard, finite-state model checking.

Since applet composition is commutative, one can apply the compositional reasoning principle also with respect to applet $\mathcal{B}$ in the second premise of the rule to yield a further decomposition of the global property.

5.2. Behavioural properties

As explained above, decomposition of global behavioural properties is more problematic, as behavioural properties in general do not give rise to unique maximal applets. We can represent the set of applets satisfying the local assumption by a model that behaviourally simulates these applets, but this necessarily leads to approximative (i.e., sound but incomplete) solutions, since such a model cannot be guaranteed to be a legal applet behaviour itself. However, by restricting local assumptions to structural properties, we obtain a complete compositional verification rule, thus avoiding the possibility of false negatives. This rule exploits the result that structural simulation implies behavioural simulation (Theorem 36).

Let $\mathcal{A} : I$ and $\mathcal{B} : J$ be applets such that $I \cup J$ is closed and let ϕ and ψ be formulae of structural and behavioural simulation logic, respectively. We propose a compositional verification principle embodied by the following rule:

$$(\mathsf{compos})\ \frac{\mathcal{A} \models_s \phi \qquad \theta_I(\phi) \uplus \mathcal{B} \models_b \psi}{\mathcal{A} \uplus \mathcal{B} \models_b \psi}\ \mathcal{A} : I$$

We establish soundness and completeness of the rule with the help of the following result, which characterises its second premise.

Proposition 38. *Let $\mathcal{B} : J$ be an applet and I an interface such that $I \cup J$ is closed. Then $\theta_I(\phi) \uplus \mathcal{B} \models_b \psi$ if and only if for all $\mathcal{A} : I$ with $\mathcal{A} \models_s \phi$ we have $\mathcal{A} \uplus \mathcal{B} \models_b \psi$.*

Proof. "⇒" Suppose $\theta_I(\phi) \uplus \mathcal{B} \models_b \psi$, $\mathcal{A} : I$ and $\mathcal{A} \models_s \phi$. Then certainly also $\mathcal{R}(\mathcal{A}) : I$ and so we get $\mathcal{A} \leqslant_s \theta_I(\phi)$ by Theorem 32(2). From Corollary 28 and Theorem 36 we derive that $\mathcal{A} \uplus \mathcal{B} \leqslant_b \theta_I(\phi) \uplus \mathcal{B}$. Hence, $\mathcal{A} \uplus \mathcal{B} \models_b \psi$ by Corollary 17. "⇐" By Theorem 32(2) we have $\theta_I(\phi) : I$ and $\theta_I(\phi) \models_s \phi$, thus $\theta_I(\phi) \uplus \mathcal{B} \models_b \psi$. □

Theorem 39. *Rule* (compos) *is sound and complete.*

Proof. Soundness is immediate by Proposition 38. For completeness suppose $\mathcal{A} \uplus \mathcal{B} \models_b \psi$ and set $\phi = \chi(\mathcal{A})$. By Theorem 9 we have $\mathcal{A} \models_b \chi(\mathcal{A})$. To establish the second premise of the rule, we use Proposition 38 and show $\mathcal{C} \uplus \mathcal{B} \models_b \psi$ for an arbitrary $\mathcal{C} : I$ with $\mathcal{C} \models_s \mathcal{X}(\mathcal{A})$. We use Theorem 9 to derive $\mathcal{C} \leqslant_s \mathcal{A}$. The result then follows by Theorem 36 and Corollaries 17 and 28. □

Again, since applet composition is commutative, one can apply the compositional reasoning principle (compos) also with respect to applet $\mathcal{B}$ in the second premise of the rule to yield a further decomposition of the global property.

Note that by taking $\mathcal{B}$ to be the empty applet $\varnothing_{J^-}$, (compos) reduces to a rule relating behavioural properties to structural ones:

$$(\mathsf{struct\text{-}beh})\ \frac{\mathcal{A} \models_s \phi \qquad \theta_I(\phi) \models_b \psi}{\mathcal{A} \models_b \psi}\ \mathcal{A} : I$$

Thus, given applet $\mathcal{A} : I$, the satisfaction of behavioural property ψ can be reduced to the satisfaction of structural property ϕ if and only if the maximal applet with respect to I and ϕ (behaviourally) satisfies property ψ.

6. Interface abstraction

So far we have only considered applets where all provided methods are public, meaning that they can be called from the outside. However, in practice the public methods will be implemented using private methods which are hidden from the outside world. Thus, when one wishes to check that an actual applet implementation (using private methods) satisfies a specified property, one needs to abstract away from the private methods, which are not observable from the outside. In particular, in a compositional verification setting, local assumptions (and global guarantees) will typically be expressed at the public interface level of an applet, while the concrete applet implementation will use private methods. For the case study presented in Section 8, the ability to distinguish between public and private method is crucial to make verification feasible.

Given an applet $\mathcal{A}$ with interface $I = (I^+, I^-)$ and a set of *public* methods $M \subseteq I^+$, we define the *public interface* of $\mathcal{A}$ by $\hat{I}(M) = (M, I^- - (I^+ - M))$. The methods in the set $I^+ - M$ are called *private* methods of $\mathcal{A}$.

We introduce the notion of *interface behaviour*, which – intuitively speaking – projects the applet behaviour onto the observable methods declared in the public interface. For the purpose of practical verification, we present the *interface abstraction* of an applet, produced by an inlining algorithm, which over-approximates the applet's interface behaviour by inlining its private methods. We also show that, under the (very common) restriction that an applet is last-call recursive, an inlined applet is weakly simulation equivalent to the interface behaviour of the original applet. We then propose a modified principle for compositional verification based on the interface abstraction of applets and the maximal model obtained for the public interface of the corresponding applet.

6.1. Interface behaviour

The next section defines an inlining algorithm that transforms a concrete applet implementation into an applet that contains only method calls to public methods. We want to prove that for any closed applet, every behaviour of the concrete applet is also a behaviour of the inlined applet. However, for this to hold, we have to abstract the concrete behaviour to the level of public methods. Therefore, we introduce the notion of *interface behaviour* of an applet with respect to a set of public methods M.

First, we define the *top* public method with respect to M, which for a given call stack σ is the first public method to which a node in the call stack belongs. For convenience, below we will often write the states of the behavioural model as a simple sequence of states, i.e., $v \cdot \sigma$, instead of (v, σ). We use reverse indexing to denote the i^{th} element from the back of a sequence, so that $(v \cdot \sigma)_{|\sigma|} = v$ (where $|\sigma|$ denotes the length of sequence σ), and σ_0 is the last element of σ. Let $\lambda_{\mathsf{Meth}}(v)$ denote the method to which node v belongs.

$$\begin{aligned} \mathsf{topindex}^M(\sigma) &= \max\{i \mid 0 \leqslant i < |\sigma| \wedge \lambda_{\mathsf{Meth}}(\sigma_i) \in M\} \\ \mathsf{top}^M(\sigma) &= \lambda_{\mathsf{Meth}}(\sigma_{\mathsf{topindex}^M(\sigma)}) \end{aligned}$$

Using these definitions, we can define a relabelling ρ^M of transition labels to the public level. Labels for calls and returns between public methods are left unchanged. A call from a private to a public method is relabelled as a call from the *top* public method in the pending call stack. A return from a public to a private method is relabelled as a return to the *top* public method. All other transitions get labelled as silent actions.

$$\rho^M((v,\sigma),\ell) = \begin{cases} \ell & \text{if } \ell = m_1\{\mathsf{call/ret}\}m_2 \wedge m_1, m_2 \in M \\ \mathsf{top}^M(\sigma)\ \mathsf{call}\ m_2 & \text{if } \ell = m_1\ \mathsf{call}\ m_2 \wedge m_1 \notin M, m_2 \in M \\ m_1\ \mathsf{ret}\ \mathsf{top}^M(\sigma) & \text{if } \ell = m_1\ \mathsf{ret}\ m_2 \wedge m_1 \in M, m_2 \notin M \\ \varepsilon & \text{otherwise} \end{cases}$$

Now we are ready to define the interface behaviour of applet $\mathcal{A}$ with respect to a set of public methods M.

Definition 40 (*Interface behaviour*)**.** Let $\mathcal{A} : I$ be a closed applet with behaviour $b(\mathcal{A}) = ((S, L, \rightarrow, A, \lambda), E)$. Let $M \subseteq I^+$ be a set of public methods. The *interface behaviour of $\mathcal{A}$ with respect to M* is defined as

$$b^M(\mathcal{A}) = \left((S, L^M, \rightarrow^M, A^M, \lambda^M), E^M\right)$$

where

- $L^M = \{m_1\, l\, m_2 \mid m_1, m_2 \in M \wedge l \in \{\mathsf{call}, \mathsf{ret}\}\} \cup \{\epsilon\}$
- $\rightarrow^M = \{((v,\sigma), \ell, (v',\sigma')) \mid \exists a \in L.\, (v,\sigma) \xrightarrow{a} (v',\sigma') \wedge \rho^M((v,\sigma), a) = \ell\}$
- $A^M = M \cup \{r\}$
- $\lambda^M = (v,\sigma) \mapsto \{\mathsf{top}^M(v \cdot \sigma)\} \cup \mathsf{if}(v \in M \wedge v \models r)\ \mathsf{then}\ \{r\}\ \mathsf{else}\ \varnothing$
- $E^M = \{v \mid v \in E \wedge \lambda_{\mathsf{Meth}}(v) \in M\}$.

The interface behaviour of an applet also defines a Pushdown Automaton.

Proposition 41. *The interface behaviour of $\mathcal{A}$ with respect to I^+ is identical to its behaviour*, i.e., $b^{I^+}(\mathcal{A}) = b(\mathcal{A})$.

We define behavioural interface simulation $\mathcal{A} \leqslant_b^M \mathcal{B}$ as $b^M(\mathcal{A}) \leqslant b^M(\mathcal{B})$, and weak behavioural interface simulation $\mathcal{A} \leqslant_{b,w}^M \mathcal{B}$ as $b^M(\mathcal{A}) \leqslant_w b^M(\mathcal{B})$. Notice that $\mathcal{A}$ and $\mathcal{B}$ need not have the same interfaces – we only require $M \subseteq I_{\mathcal{A}}^+$ and $M \subseteq I_{\mathcal{B}}^+$. Similarly, for any formula ϕ in simulation logic over L^M and A^M, we define behavioural interface satisfaction $\mathcal{A} \models_b^M \phi$ as $b^M(\mathcal{A}) \models \phi$, and weak behavioural interface satisfaction $\mathcal{A} \models_{b,w}^M \phi$ as $b^M(\mathcal{A}) \models_w \phi$.

6.2. The inlining transformation

Next we define an inlining algorithm α_M that, given a set of public methods M, transforms an applet graph by inlining all private calls. Recursive calls to private methods are not inlined, but create loops in the resulting graph. We prove that the interface behaviour of the original applet $\mathcal{A}$ is simulated by the behaviour of the inlined applet $\alpha_M(\mathcal{A})$, thus (by Corollary 17) all properties ϕ of the latter, i.e., $\alpha_M(\mathcal{A}) \models_b \phi$, are also properties of the former, i.e., $\mathcal{A} \models_b^M \phi$. Moreover, we prove that if the applet is last-call recursive, the two behaviours are weakly simulation equivalent – thus both applets satisfy exactly the same observable properties at the public interface level.

Notice that the inlining algorithm does not require the applet to be closed: it treats all external methods as public.

6.2.1. The inlining algorithm

The algorithm is applied to each public method and (recursively) inlines all calls to private methods. Intuitively, constructing the transformed (or inlined) graph for a public method m corresponds to executing the interface behaviour of m, where method calls to public methods are skipped and recursion is replaced by iteration. The nodes of the inlined applet can thus be seen as states of the (interface) behaviour of the original applet, modulo an abstraction function which replaces recursion by iteration.

During the inlining, each edge that represents internal transfer or a call to a public method is left unchanged. Each edge that represents a call to a private method is replaced by two internal edges: one from the calling point to the entry point of the method; and another from the return point of the method to the destination of the calling edge. If a method has several entry or return points, several internal edges are created. The private method is inlined recursively. Each node is replaced by a sequence denoting the fragment of the call stack from the activation of the public method up to the current node (except for the case of a recursive call). Since we keep track of the pending call stack, we can recognise recursive calls to private methods. In that case, the appropriate initial fragment of the call stack is used to decide the exact new edges.

For the formal definition of the inlining algorithm, we need some new notions. Let $\mathcal{A} : I$ be an applet and $M \subseteq I^+$ be a set of public methods. An *M-frame* is a sequence of nodes σ of which only $\lambda_{\mathsf{Meth}}(\sigma_0)$ is in M. An M-frame is called *normal*, if all nodes in the frame belong to different methods. The nodes of the inlined applet are represented by normal M-frames derived from the behaviour of the original applet. The abstraction function mentioned above (replacing recursion by iteration) is formalised by means of the (normalising) conditional rewrite rule $\sigma \cdot v \cdot \sigma' \cdot v' \cdot \sigma'' \hookrightarrow \sigma \cdot v \cdot \sigma''$ if $\lambda_{\mathsf{Meth}}(v) = \lambda_{\mathsf{Meth}}(v')$ and $\sigma' \cdot v' \cdot \sigma''$ is a normal M-frame. Let $\nu(\sigma)$ denote the normal form of σ with respect to the rule. Note that if σ is an M-frame, then $\nu(\sigma)$ is a normal M-frame. Moreover, for any M-frame σ we have $\mathsf{top}^M(\sigma) = \lambda_{\mathsf{Meth}}(\sigma_0)$.

Further, for method m we define $\mathcal{I}nt(m)$ and $\mathcal{C}all(M, m)$, denoting the sets of its internal edges and call edges with respect to methods in a set M, respectively.

$$\begin{aligned} \mathcal{I}nt(m) &= \{(v, \varepsilon, v') \mid v \to_m v' \wedge v \models \neg r\} \\ \mathcal{C}all(M, m) &= \{(v, m', v') \mid v \xrightarrow{m'}_m v' \wedge v \models \neg r \wedge m' \in M\} \end{aligned}$$

The definition of the inlining algorithm uses auxiliary functions η and ζ. The function η considers all edges related to a method: it returns internal and public call edges with renamed nodes – using the pending call stack, and calls function ζ on private call edges. Function ζ adds edges to the entry point, and from the return point of the private method, using the pending call stack argument, and if necessary normalising the result (this uses the fact that the pending call stack is always a normalised M-frame). Then it checks if the private call is non-recursive, in which case the private method is inlined recursively.

Definition 42 (*Inlined applet*)**.** Let $\mathcal{A} : I$ be an applet, and let (M, P) be a partitioning of I^+ into public and private methods, respectively. We define the *inlined applet*

$$\alpha_M(\mathcal{A}) = ((V', L', \to', A', \lambda'), E')$$

where

- $V' = \{w \in V^+ \mid w \text{ is a normal } M\text{-frame}\}$,
- $L' = (I^- - P) \cup \{\varepsilon\}$,
- $\to' = \bigcup_{m \in M} \eta(m, \epsilon)$ *where*

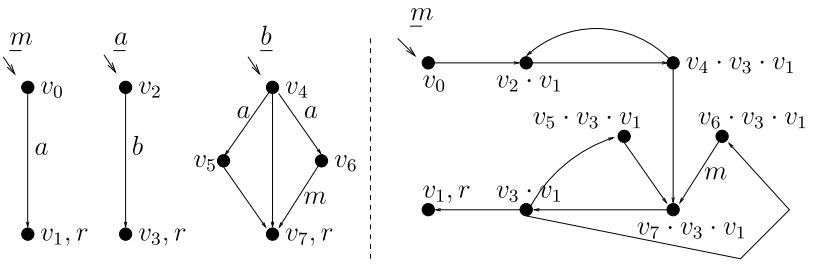

Fig. 5. Example applet before and after inlining.

$$
\begin{array}{rcl}
\eta(m,\sigma) & = & \{(v\cdot\sigma,\ell,v'\cdot\sigma) \mid (v,\ell,v') \in \mathcal{I}nt(m) \cup \mathcal{C}all(I^- - P, m)\} \\
 & \cup & \bigcup\{\zeta(\sigma,(v,m',v')) \mid (v,m',v') \in \mathcal{C}all(P,m)\} \\
\zeta(\sigma,(v,m',v')) & = & \{(v\cdot\sigma,\varepsilon,\nu(e\cdot v'\cdot\sigma)) \mid e \models m' \wedge e \in E\} \\
 & \cup & \{(\nu(rt\cdot v'\cdot\sigma),\varepsilon,v'\cdot\sigma) \mid rt \models (m' \wedge r)\} \\
 & \cup & \text{if } \neg\exists i.\,(0 \leqslant i \leqslant |\sigma| \wedge (v'\cdot\sigma)_i \models m') \\
 & & \text{then } \eta(m',v'\cdot\sigma) \text{ else } \varnothing
\end{array}
$$

- $A' = M \cup \{r\}$
- $\lambda' = \sigma \mapsto \{\lambda_{\mathsf{Meth}}(\sigma_0)\} \cup \text{if } (|\sigma| = 1 \wedge \sigma_0 \models r) \text{ then } \{r\} \text{ else } \varnothing$
- $E' = \{v \in E \mid \lambda_{\mathsf{Meth}}(v) \in M\}$.

Before discussing properties of the inlining algorithm, we first show an example.

Example 43. Suppose we have an applet as depicted in the left-hand column of Figure 5. Inlining this applet with the public method set $\{m\}$ results in the applet depicted in the right-hand column of Figure 5. Notice that all internal and public call edges are preserved, while private method calls are replaced by two edges: to the entry and from the return point of the called method, respectively.

6.2.2. Properties

We state several useful properties of the inlining algorithm. First of all, the inlining algorithm computes an applet having as interface the public interface of the original applet.

Proposition 44. *Let $\mathcal{A} : I$ be an applet and $M \subseteq I^+$ a set of public methods. Then $\alpha_M(\mathcal{A})$ is an applet with interface $\hat{I}(M)$,* i.e., $\alpha_M(\mathcal{A}) : \hat{I}(M)$.

By Proposition 41 we thus get:

$$b^M(\alpha_M(\mathcal{A})) = b(\alpha_M(\mathcal{A}))$$

Since the inlining transformation α_M only inlines provided methods not in M, α_{I^+} is the identity operation.

Proposition 45. *Let $\mathcal{A} : I$ be an applet. Then $\alpha_{I^+}(\mathcal{A}) = \mathcal{A}$.*

Finally, the inlining algorithm enjoys the following distributivity property.

Proposition 46. *Let $\mathcal{A} : I_\mathcal{A}$ and $\mathcal{B} : I_\mathcal{B}$ be applets such that $I^+_\mathcal{A}$ and $I^+_\mathcal{B}$ are disjoint and let $M_\mathcal{A} \subseteq I^+_\mathcal{A}$ and $M_\mathcal{B} \subseteq I^+_\mathcal{B}$ be sets of public methods such that $I^-_\mathcal{A} \subseteq I^+_\mathcal{A} \cup M_\mathcal{B}$ and $I^-_\mathcal{B} \subseteq I^+_\mathcal{B} \cup M_\mathcal{A}$. Then*

$$\alpha_{M_\mathcal{A} \cup M_\mathcal{B}}(\mathcal{A} \uplus \mathcal{B}) = \alpha_{M_\mathcal{A}}(\mathcal{A}) \uplus \alpha_{M_\mathcal{B}}(\mathcal{B})$$

6.2.3. Simulation results

As already mentioned, the interface behaviour of the original applet is over-approximated by the inlining algorithm, i.e., every execution of the interface behaviour of $\mathcal{A}$ is an execution of the behaviour of $\alpha_M(\mathcal{A})$. This is due to the close correspondence between the interface behaviour of $\mathcal{A}$ and the structure of $\alpha_M(\mathcal{A})$. We provide an "inlining" transformation α'_M on the states of $b^M(\mathcal{A})$ by defining $\alpha'_M(v,\sigma) = \big(hd(\gamma), tl(\gamma)\big)$, where $\gamma = \beta_M(v\cdot\sigma)$ and where $\beta_M(\sigma)$ denotes the sequence of normalised M-frames. Notice that we always have $hd(hd(\gamma)) = hd(v\cdot\sigma)$. We show that α'_M is a simulation relating the original interface behaviour with the inlined behaviour.

Theorem 47. *Let $\mathcal{A} : I$ be a closed applet, and let $M \subseteq I^+$. Then $b^M(\mathcal{A}) \leqslant b(\alpha_M(\mathcal{A}))$.*

Proof. We show by co-induction that α'_M is a simulation between $b^M(\mathcal{A})$ and $b(\alpha_M(\mathcal{A}))$, i.e., we show that (1) the valuations of (ν,σ) in $b^M(\mathcal{A})$ and $\alpha'_M(\nu,\sigma)$ in $b(\alpha_M(\mathcal{A}))$ agree, and (2) if $(\nu,\sigma) \xrightarrow{l} (\nu',\sigma')$ in $b^M(\mathcal{A})$, then we have $\alpha'_M(\nu,\sigma) \xrightarrow{l} \alpha'_M(\nu',\sigma')$ in $b(\alpha_M(\mathcal{A}))$. The result then follows since α'_M maps the entry states of $b^M(\mathcal{A})$ to entry states of $b(\alpha_M(\mathcal{A}))$ (in fact, the entry states coincide, and α'_M maps every entry state to itself). It is easy to check that the valuations agree and that the transitions are simulated. For the full proof we refer to our technical report [37]. □

Notice that in general we do not have behavioural simulation equivalence. The inlining construction introduces transfer edges for calls to and returns from private methods. Because of the latter, the behaviour of the inlined applet can contain a silent transition corresponding to a return from a private method in the original applet, even when the inlined applet has not yet followed a silent transition corresponding to a call to this private method in the original applet. For instance, the execution $(v_0,\varepsilon) \to (v_2.v_1,\varepsilon) \to (v_4.v_3.v_1,\varepsilon) \to (v_7.v_3.v_1,\varepsilon) \to (v_3.v_1,\varepsilon) \to (v_6.v_3.v_1,\varepsilon) \xrightarrow{m\,\mathsf{call}\,m} (v_0, v_7.v_3.v_1)$ of the inlined applet in Figure 5 does not correspond to any execution in the original applet. The inlining transformation thus introduces new behaviours. Notice however, that these new behaviours are only observable in applets which are not last-call recursive.

A set of methods is *recursive* if every method in the set contains a (reachable) call edge to some method in the set. A call edge is recursive if the calling and the called methods belong to some minimal (and thus, mutually) recursive method set. A program is called *last-call recursive* if from any destination node of any recursive call edge, only transfer edges are reachable. In addition, we shall assume that a return node is reachable from every such destination node.

For last-call recursive applets, we prove the reverse correspondence for observable behaviours.

Theorem 48. *Let $\mathcal{A} : I$ be a closed last-call recursive applet, and let $M \subseteq I^+$. Then $b(\alpha_M(\mathcal{A})) \leqslant_w b^M(\mathcal{A})$.*

Proof. Consider a state (w,γ) in $b(\alpha_M(\mathcal{A}))$, where $\lambda_{\mathsf{Meth}}(hd(w)) \notin M$ and $hd(w) \models r$. For last-call recursive applets, the inlining transformation α_M has the property that for any such w, the nodes w' such that $\nu(hd(w)\cdot w') = w$ but $hd(w)\cdot w' \neq w$ and which are structurally reachable from w in $\alpha_M(\mathcal{A})$ form (together with w) a strongly connected component and are equivalent with respect to structural simulation. As a consequence, in $b(\alpha_M(\mathcal{A}))$, all states (w',γ) for a given γ also form a strongly connected component and are weak simulation equivalent. Modulo such "return" equivalence classes, we show by co-induction that $(\alpha'_M)^{-1}$ is a weak simulation between $b(\alpha_M(\mathcal{A}))$ and $b^M(\mathcal{A})$. More exactly, we show that (1) the valuations of $\alpha'_M(\nu,\sigma)$ and (ν,σ) agree, and (2) if $\alpha'_M(\nu,\sigma) \xrightarrow{l} (w',\gamma')$ is a transition in $b(\alpha_M(\mathcal{A}))$ other than a (silent) transition within a return equivalence class, then $(\nu,\sigma) \stackrel{l}{\Rightarrow} (\nu',\sigma')$ in $b^M(\mathcal{A})$ for some ν' and σ' such that $\alpha'_M(\nu',\sigma') = (w',\gamma')$ (in most cases we even show the corresponding strong transition). The result then follows since α'_M maps entry states of $b(\alpha_M(\mathcal{A}))$ to entry states of $b^M(\mathcal{A})$. It is easy to check that the valuations agree and that the transitions are simulated. For the full proof we again refer to [37]. □

Since weak simulation contains simulation we have the following.

Corollary 49. *Let $\mathcal{A} : I$ be a closed last-call recursive applet, and let $M \subseteq I^+$. Then $b^M(\mathcal{A}) \simeq_w b(\alpha_M(\mathcal{A}))$.*

6.3. Interface abstraction and compositional reasoning

Using the results obtained above, we can state several verification principles that can be used to prove properties of applet interface behaviour. We first present two abstraction principles, and then show how these can be combined with our compositional verification principle from Section 5.

6.3.1. Abstraction rules

Let $\mathcal{A} : I$ be a closed applet, and let $M \subseteq I^+$. With the results established above, we can justify the following abstraction principle (abstract), where ψ is a behavioural interface formula.

$$\text{(abstract)}\ \frac{\alpha_M(\mathcal{A}) \models_b \psi}{\mathcal{A} \models_b^M \psi}$$

Theorem 50. *Rule* (abstract) *is sound.*

Proof. Follows from the definition of behavioural satisfaction, Theorem 47, Corollary 17, and the definition of behavioural interface satisfaction. □

When $\mathcal{A}$ is last-call recursive, we can even provide a faithful abstraction principle (weak-abstract) for properties of the observable behaviour by using transformation δ mentioned in Section 3.4.

$$\text{(weak-abstract)}\ \frac{\alpha_M(\mathcal{A}) \models_b \delta(\psi)}{\mathcal{A} \models^M_{b,w} \psi}$$

Theorem 51. *For last-call recursive applets $\mathcal{A}$ rule* (weak-abstract) *is sound and complete.*

Proof. Follows from the definition of behavioural satisfaction, Proposition 20(ii), Corollary 49, Corollary 23, and the definition of weak behavioural interface satisfaction, all of which are equivalences. □

6.3.2. Compositional reasoning

Let $\mathcal{A} : I_\mathcal{A}$ and $\mathcal{B} : I_\mathcal{B}$ be applets such that $I_\mathcal{A} \cap I_\mathcal{B} = \varnothing$ and let $M_\mathcal{A} \subseteq I^+_\mathcal{A}$ and $M_\mathcal{B} \subseteq I^+_\mathcal{B}$ be sets of public methods such that $I^-_\mathcal{A} \subseteq I^+_\mathcal{A} \cup M_\mathcal{B}$ and $I^-_\mathcal{B} \subseteq I^+_\mathcal{B} \cup M_\mathcal{A}$. The latter condition says that each applet only calls its own methods and the others' public methods and implies that their composition is closed. We combine the compositional verification principle (compos) from Section 5 with the abstraction principle (abstract) to obtain the following abstract compositional verification principle:

$$\text{(abstract-compos)}\ \frac{\alpha_{M_\mathcal{A}}(\mathcal{A}) \models_s \phi \qquad \theta_{\hat{I}(M_\mathcal{A})}(\phi) \uplus \alpha_{M_\mathcal{B}}(\mathcal{B}) \models_b \psi}{\mathcal{A} \uplus \mathcal{B} \models_b^{M_\mathcal{A} \cup M_\mathcal{B}} \psi}$$

Notice that the maximal model construction is based on the public interface $\hat{I}(M_\mathcal{A}) = (M_\mathcal{A}, I^-_\mathcal{A} - (I^+_\mathcal{A} - M_\mathcal{A}))$ of applet $\mathcal{A}$.

Theorem 52. *Rule* (abstract-compos) *is sound.*

Proof. Follows from the soundness of (abstract) and (compos) together with Proposition 46. □

Similarly as for the abstraction principle, we can state a faithful compositional verification principle (weak-abstract-compos) for properties of the observable interface behaviour of applets which are last-call recursive:

$$\text{(weak-abstract-compos)}\ \frac{\alpha_{M_\mathcal{A}}(\mathcal{A}) \models_s \phi \quad \theta_{\hat{I}(M_\mathcal{A})}(\phi) \uplus \alpha_{\mathcal{M}_\mathcal{B}}(\mathcal{B}) \models_b \delta(\psi)}{\mathcal{A} \uplus \mathcal{B} \models^{M_\mathcal{A} \cup M_\mathcal{B}}_{b,w} \psi}$$

Theorem 53. *Rule* (weak-abstract-compos) *is sound and complete for last-call recursive applets $\mathcal{A}$ and $\mathcal{B}$.*

Notice that rule (weak-abstract-compos) is also sound for applets that are not last-call recursive: last-call recursiveness is only needed to ensure completeness.

Our scenario for secure post-issuance loading of applets is based on the verification principle embodied by these rules and its derivatives. In particular, a combined application of rules (weak-abstract-compos) and (compos) yields the rule (w(eak)-a(bstract)-c(ompos)-2), which we apply in our case study in Section 8:

$$\text{(wac-2)}\ \frac{\alpha_{M_\mathcal{A}}(\mathcal{A}) \models_s \phi \qquad \alpha_{M_\mathcal{B}}(\mathcal{B}) \models_s \xi \qquad \theta_{\hat{I}(M_\mathcal{A})}(\phi) \uplus \theta_{\hat{I}(M_\mathcal{B})}(\xi) \models_b \delta(\psi)}{\mathcal{A} \uplus \mathcal{B} \models^{M_\mathcal{A} \cup M_\mathcal{B}}_{b,w} \psi}$$

Here, an application of rule (compos) has introduced a second maximal model for the public interface of $\mathcal{B}$ and structural property ξ. Notice that this rule is sound *and* complete for last-call recursive applets.

7. A tool set for compositional verification

To support our compositional verification method, we have developed a tool set implementing the various algorithms presented above and providing translations into the input formats of appropriate, existing model checkers. Figure 6 gives a general overview of the tool set.

As input we have for each applet either an implementation (in Java bytecode), or a structural property, restricting its possible implementations, plus a public interface, specifying the methods provided and required by the applet. For these inputs, we construct an applet representation according to Definition 27.

In case we have the applet implementation, we use the *Applet Analyser* to extract the concrete applet graph. The Applet Analyser is a static analysis tool, built on top of the SOOT Java Optimisation Framework [12]. The byte code of an applet is transformed into Jimple basic blocks, while abstracting away variables, method parameters, and calls to API methods. We use SOOT's standard class hierarchy analysis to produce a safe over-approximation of the call graph. If, for example, the static analysis cannot determine the receiver of a virtual method call, a call edge is generated for every possible method implementation. Next we use the *Inliner,* which is an Ocaml implementation of the inlining algorithm of Definition 42. The Inliner takes the extracted method graph and the public interface as input, and produces the graph at the public interface level.

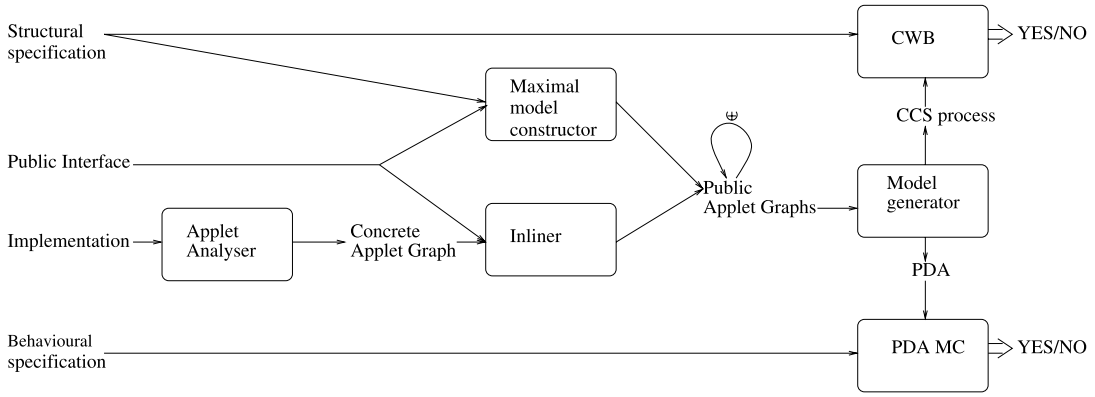

Fig. 6. Tool set for compositional applet verification.

In case we have a structural property, we use the *Maximal Model Constructor*. This is an Ocaml implementation of the SNF transformation as defined in Section 3.3, which we use to construct maximal models. The structural properties and the applet interface are used to produce an applet graph that simulates all possible implementations of applets respecting the formula.

If required, the resulting applets can be composed with the $\uplus$ operator, which is basically a concatenation of the textual graph representations. Since the applet analyser appends package names to the method names, there are no name conflicts to be resolved here. Using the *Model Generator* the resulting applet graphs are translated into models which serve as input for different model checkers. If we want to check structural properties, we exploit the fact that applet graphs can be viewed as finite Kripke structures. Therefore, structural properties can be expressed in temporal logics and they can be checked using standard model checking tools such as the Concurrency Workbench (CWB) [38]. The Kripke structures of the CWB are labelled transition systems generated from CCS process definitions. For this purpose, we use the Model Generator to convert applet graphs into a representation as CCS processes. Since CCS does not have the notion of valuation, atomic propositions p assigned to a node in an applet are represented by *probes*, that is, self-loops labelled by p. The translation also produces a set of process constants corresponding to the entry nodes of the respective applet. To model check an applet graph against a structural safety property, all initial states have to be checked individually. We encode the properties to be checked as μ-calculus formulae, replacing atomic propositions p by $\langle p \rangle \mathsf{true}$. Since CWB supports parametrised formulae, our specification patterns can be encoded directly.

If for a composed system we want to verify whether it respects a behavioural safety property, we use the fact that the behaviour of an applet is an infinite state model generated by a Pushdown Automaton (PDA) given as a set of production rules induced by the applet. The model checking problem for this class of models is exponential both in the size of the formula and in the number of control states of the PDA [10]. Ideally we would like to use an existing model checker for PDAs (PDA MC). Unfortunately, there is currently no efficient tool available for model checking (alternation-free) modal μ-calculus properties of PDAs. We experimented with Alfred [39], a demonstrator tool implementing the model checking algorithm of Bouajjani et al. [40], and we are currently developing such a model checker.

8. Case study

To evaluate its validity, we apply our compositional verification method to a realistic smart card case study, which illustrates typical unwanted applet interactions. The application, an electronic purse, has been provided by smart card producer Gemplus as a test case for formal methods. Even though it is not actually used by Gemplus, it demonstrates all the relevant issues related to smart card applications. In this section, we introduce the electronic purse case study, present the local and global specifications for the different applets, and describe their verification using the tool set presented above.

8.1. Illicit applet interactions in the electronic purse

The Gemplus electronic purse case study PACAP [41] is developed to provide a realistic case study for applying formal methods to Java Card applications. It defines three applications: *CardIssuer*, *Purse* and *Loyalty*. Typically, a card will contain one card issuer and one purse applet, but several loyalty applets. The case study has been previously used in connection with several other formal techniques. For example, functional source code level specifications have been given and checked with automatic and interactive verification techniques [42]. The case study also has been used to illustrate an approach where different privacy levels are assigned to information, and model checking is used to ensure that the information flow respects the restrictions imposed by these privacy levels [43]. The property described in the latter work motivates the property we

study here. However, our technique is more general, allowing the verification of arbitrary behavioural control-flow safety properties.

The property that we verify for this case study is only concerned with *Purse* and *Loyalty*, we shall therefore not discuss *CardIssuer* any further. If the card holder wishes to join a loyalty program, the appropriate applet can be loaded on the card. Subsequently, the purse and the different loyalties will exchange information about the purchases made, so that the loyalty points can be credited. Current versions of Java Card use sharable interfaces to exchange this kind of information. Even though in the future this is likely to change, for our techniques to be applicable it is not relevant how this communication exactly takes place, as long as it is in terms of method calls (rather than in terms of shared state). The goal of our case study is to ensure that no illicit interactions can happen between the various applets on the card. The code of the application is last-call recursive, thus our verification will be exact, and the inlining step will not introduce any new observable interface behaviours. In this particular case study, we can verify correctness of the decomposition, thus we rely only on soundness of the compositional verification principle. However, if correctness of the decomposition could not be verified, the completeness for last-call recursive applets would tell us that our local assumption is too weak.

To understand the property which we verify here, let us look closer at how the purse and the loyalties communicate about the purchases made with the card. The electronic purse keeps a log table of all credit and debit transactions, and the loyalty applets can request the (relevant) information stored in this table. Further, loyalties might have so-called partner loyalties, which means that a user can add up the points obtained with the different loyalty programs. Therefore, each loyalty should keep track of its local balance and its extended balance. If the user wishes to know how many loyalty points are available exactly, the loyalty applet will ask for the relevant entries of the purse log table in order to update its balance, and it will also ask the balances of partner loyalties in order to compute the extended balance.

For efficiency reasons, the log table is of fixed length, arranged as a ring. If the log table is full, existing entries will be replaced by new transactions. In order to ensure that loyalties do not miss any of the logged transactions, they can subscribe to the so-called *logFull* service. This service signals all subscribed loyalties that the log table will be overwritten soon, and that therefore they should update their balances. Typically, loyalties will have to pay for this service.

Suppose we have an electronic purse, which contains besides the electronic purse itself two partner loyalties, say L_1 and L_2. Further, suppose that L_1 has subscribed to the *logFull* service, while L_2 has not. If in reaction to the *logFull* message L_1 always calls an interface method of L_2 (say to ask for its balance when computing the extended balance), L_2 can implicitly deduce that the log table might be full. A malicious implementation of L_2 might therefore request the information stored in the log table before returning the value of its local balance to L_1. If loyalties have to pay for the *logFull* service, such control flow is unwanted, since the owner of the *Purse* applet will not want other loyalties to get this information for free.

This is a typical example of an illicit applet interaction, that our compositional verification technique can detect. Below, we show how the absence of this particular undesired scenario can be specified and verified algorithmically. We use compositional reasoning to reduce the global behavioural property expressing the absence of the scenario described above to local structural properties of the purse and loyalty applet classes. We assume there is only one purse applet on the card, but we allow an arbitrary number of loyalty applets on the card. However, since all loyalty applets have the same interface, we can apply class-based analysis, and treat all loyalty instances in a similar way. The case study provides implementations for the purse and the loyalty applet. These are checked against the corresponding structural properties. Notice that a typical use of the card initially only will have the purse applet installed on the card. After the card has been issued, new loyalty applets will be installed whenever the card holder wishes to join a loyalty program. Every time a new loyalty applet is installed, it will have to be verified against the structural specification of the loyalty applet.

8.2. Specification

This section presents the formalisation of the global and local security properties that we need for our example. The following section then shows how the tool set is used for the verification of the decomposition and of the implementations with respect to the local properties.

8.2.1. Specification patterns

Since writing specifications in the modal μ-calculus is known to be difficult (even in the simulation logic fragment), we define a collection of commonly used specification patterns (inspired by the Bandera Specification Pattern project [44]). In our experience, all relevant behavioural control-flow safety properties can be expressed using a small set of such patterns – however, it is important to remember that one can always fall back on the full expressiveness of simulation logic. We present several specification patterns, both at structural and behavioural level, which are all used in the case study at hand. From now on we shall adopt the convention of denoting structural properties by σ and behavioural ones by ϕ.

Structural specification patterns We shall use *Everywhere* with the obvious formalisation:

$$\begin{aligned} \mathit{Everywhere}\ \sigma &= \nu Y.\, \sigma \wedge [\varepsilon, I^-]Y \\ &= Y[Y = \sigma \wedge [\varepsilon, I^-]Y] \end{aligned}$$

as well as the following patterns, for method sets M and M' of an applet with interface I:

$$\begin{array}{rcl} M\ \mathit{HasNoCallsTo}\ M' & = & (\bigwedge_{m\in M} \neg m) \vee (\mathit{Everywhere}\ [M']\,\mathsf{ff}) \\ \mathit{HasNoOutsideCalls}\ M & = & M\ \mathit{HasNoCallsTo}\ (I^- - M) \end{array}$$

The first pattern specifies that method graphs in the set M do not contain edges labelled with elements of the set M'. The second specifies a closed set of methods M, i.e., methods in M only contain calls to methods in M.

Behavioural specification patterns Pattern *Always* is standard:

$$\begin{array}{rcl} \mathit{Always}\ \phi & = & \nu Z.\ \phi \wedge [L_b]Z \\ & = & Z[Z = \phi \wedge [L_b]Z] \end{array}$$

For specifying that a property ϕ is to hold within a call to method m, we use the *Within* pattern formalised as follows:

$$\mathit{Within}\ m\ \phi\ =\ \neg m \vee (\mathit{Always}\ \phi)$$

More precisely, this pattern states that ϕ always holds as soon as m is called. However, since we do not use this pattern inside other formulae, the given description is correct. Notice that this is a typical behavioural pattern: the notion of *Within* a method invocation encompasses all methods that might be invoked during the call to m. This reachability notion cannot be directly expressed at the structural level.

Finally, for applet $\mathcal{A} : (I^+, I^-)$ and method set M, we define:

$$\mathit{CanNotCall}\ \mathcal{A}\ M\ =\ \bigwedge_{m\in I^+} \bigwedge_{m'\in M} [m\ \mathit{call}\ m']\mathsf{ff}$$

This pattern holds for state (v, σ) if no call to a method in M is possible.

8.2.2. The security properties

We express the security properties at the *public* level, that is, structural properties refer to the interface abstraction (i.e., inlined version) and behavioural properties to the interface behaviour of applets. As mentioned above, communication between applets takes place via so-called sharable interfaces. The *Purse* applet defines a sharable interface SI_P for communication with loyalty applets, containing the methods *getTransaction, isThereTransaction*, *getInvExchangeRateIntPart* and *getInvExchangeRateDecPart.* The *Loyalty* applet defines two sharable interfaces: one, SI_{LP}, for communication with a *Purse*, containing the methods *logFull* and *exchangeRate,* and one, SI_{LL}, for communication with other loyalty applets, containing methods *getBalance* and *getDebit*. If we define $SI_L = SI_{LP} \cup SI_{LL}$, then we can identify the following public interfaces: $I_P = (SI_P, SI_P \cup SI_L)$ for *Purse*, and $I_L = (SI_L, SI_P \cup SI_L)$ for *Loyalty.*

The global security property To guarantee that no loyalty will get the opportunity to circumvent subscribing to the *logFull* service, we require that if the *Purse* calls the *logFull* method of a loyalty, within this call the loyalty does not communicate with other loyalties. However, as the *logFull* method is supposed to call the *Purse* for its transactions, we also have to exclude indirect communications, via the *Purse*. We require the following global property of the interface behaviour:

> A call to *Loyalty.logFull* does not trigger any calls to any other loyalty.

This property can be formalised with the help of behavioural patterns:

$$\begin{array}{rcl} \phi & = & \mathit{Within}\ \mathit{Loyalty.logFull} \\ & & (\mathit{CanNotCall}\ \mathit{Loyalty}\ SI_L\ \wedge \mathit{CanNotCall}\ \mathit{Purse}\ SI_L) \end{array}$$

Thus, if a loyalty receives a *logFull* message, it cannot call any other loyalty (because it cannot call any of its sharable interface methods), and in addition, if the *Purse* is (re)activated within the call to *logFull,* it cannot call any loyalty applet.

Property decomposition We apply rule (wac-2) from Section 6.3 and therefore introduce local structural properties for the inlined versions of *Purse* and *Loyalty*. Here we explain the formalisation of the local properties; below we describe how we actually verify that these are sufficient to guarantee the global behavioural property. Within *Loyalty.logFull,* the *Loyalty* applet has to call the methods *Purse.isThereTransaction* and *Purse.getTransaction*, but it should not make any other external calls (where calls to sharable interface methods of *Loyalty* are considered external). Notice that since we are performing class-based analysis, we cannot distinguish between calls to interface methods of other instances, and those of the same instance. Thus, a natural structural property for *Loyalty* would be, informally:

> From any entry point of *Loyalty.logFull*, the only reachable external calls are calls to *Purse.isThereTransaction* and *Purse.getTransaction*.

Thus, within a call to *Loyalty.logFull* the *Purse* applet can only be activated via *Purse.isThereTransaction* or *Purse.getTransaction*. For *Purse* we can therefore propose the following informal structural property:

> From any entry point of *Purse.isThereTransaction* or *Purse.getTransaction*, no edge labelled by an external call is reachable.

Using the structural specification patterns, we can specify these properties as follows.

$$\begin{array}{lcl}\sigma_L & = & \{\mathit{Loyalty.logFull}\}\ \mathit{HasNoCallsTo}\\ & & (SI_P \cup SI_L) - \{\mathit{Purse.isThereTransaction}, \mathit{Purse.getTransaction}\}\\ \sigma_P & = & \mathit{HasNoOutsideCalls}\ \{\mathit{Purse.isThereTransaction}\} \wedge\\ & & \mathit{HasNoOutsideCalls}\ \{\mathit{Purse.getTransaction}\}\end{array}$$

Notice that these specifications are expressed with respect to the inlined versions of the applets. Excluding external calls from a method at the public level is equivalent to excluding external calls from any private method that can be called transitively from the public method at the implementation level - a property which is not directly expressible (at the implementation level) in our logic (cf. Huisman et al. [30]).

8.3. Verification

After the global and local security properties have been specified, we have to show that: (1) the local properties are sufficient to establish the global security property, and (2) the implementations of the different applets respect the local properties. In order to do this, we identify the following (independent) tasks, considered in detail below.

(1) Verifying the correctness of the property decomposition by:

(*a*) building $\theta_{I_P}(\sigma_P)$ and $\theta_{I_L}(\sigma_L)$, the maximal applets for σ_P and σ_L, respectively; and
(*b*) model checking $\theta_{I_P}(\sigma_P) \uplus \theta_{I_L}(\sigma_L) \models_b \delta(\phi)$.

(2) Verifying the local structural properties by:

(*a*) extracting the applet graphs P of the *Purse* and L of the *Loyalty*;
(*b*) computing $\alpha_{SI_P}(P)$ and $\alpha_{SI_L}(L)$ using the inlining algorithm; and
(*c*) model checking $\alpha_{SI_P}(P) \models_s \sigma_P$ and $\alpha_{SI_L}(L) \models_s \sigma_L$.

We then apply rule (wac-2) to conclude that $P \uplus L \models_{b,w}^{SI_P \cup SI_L} \phi$ as required.

8.3.1. Verification of the property decomposition

To illustrate the procedure of constructing a maximal applet, we present in some detail the construction of the maximal applet for σ_L; for σ_P the construction is similar. First, we write σ_L as a modal equation system, where we use *lf* to abbreviate *Loyalty.logFull, gT* for *Purse.getTransaction, iTT* for *Purse.isThereTransaction,* and M for $(SI_P \cup SI_L) - \{iTT, gT\}$:

$$\sigma_L = \neg lf \vee Y[Y = [M]\mathsf{ff} \wedge [\varepsilon, gT, iTT]Y]$$

Next, we build the interface formula ϕ_{I_L} for interface I_L (recall that the maximal applet for σ_L is the maximal model for $\sigma_L \wedge \phi_{I_L}$). For clarity of presentation we shall make here the simplifying assumption that $SI_L = \{lf\}$; the actual case study has naturally been performed for the full sharable interface. Thus $\phi_{I_L} = X_{lf}[X_{lf} = [\varepsilon, lf, SI_P]X_{lf} \wedge lf]$. We then form the conjunction $\sigma_L \wedge \phi_{I_L}$, which by introducing a new variable Z yields:

$$Z\begin{bmatrix} Z & = & (\neg lf \vee Y) \wedge X_{lf}\\ Y & = & [M]\mathsf{ff} \wedge [\varepsilon, gT, iTT]Y\\ X_{lf} & = & [\varepsilon, lf, SI_P]X_{lf} \wedge lf\end{bmatrix}$$

The next step is to transform this formula into SNF. First, in Phase I of the transformation, each equation is transformed into a disjunction of state normal forms. Suppose we start with the equation defining Z.

(1) Make equation strongly guarded, by rewriting with the original equations:

$$Z = (\neg lf \vee ([M]\mathsf{ff} \wedge [\varepsilon, gT, iTT]Y)) \wedge [\varepsilon, lf, SI_P]X_{lf} \wedge lf$$

(2) Put equation into DNF and simplify:

$$Z = [M]\mathsf{ff} \wedge [\varepsilon, gT, iTT]Y \wedge [\varepsilon, lf, SI_P]X_{lf} \wedge lf$$

(3) Group and complete boxes. No boxes are missing, thus we only group them (remember $M = (SI_P \cup SI_L) - \{gT, iTT\} = (SI_P \cup \{lf\}) - \{gT, iTT\}$):

$$Z = [M]\mathsf{ff} \wedge [\varepsilon, gT, iTT](Y \wedge X_{lf}) \wedge lf$$

(4) Introduce new equations for formulae under boxes. Since the map h does not yet contain an entry for $\{Y, X_{lf}\}$, we choose a fresh variable U and add $(\{Y, X_{lf}\}, U)$ to h. The equation defining Z becomes

$$Z = [M]\mathsf{ff} \wedge [\varepsilon, gT, iTT]U \wedge lf$$

while we introduce the new equation $U = Y \wedge X_{lf}$.

(5) Finally, complete the equation by adding missing literals and put the formula into DNF again. Here, only literal r is missing. Adding this gives:

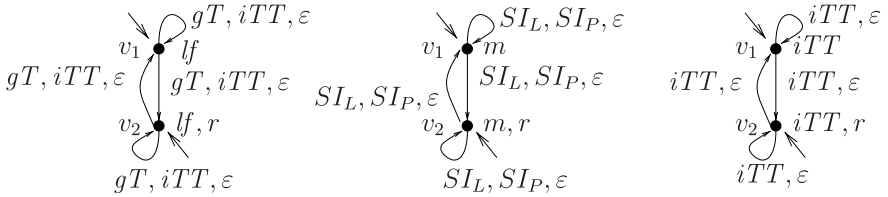

Fig. 7. Maximal applets for σ_L and σ_P.

$$\begin{array}{lcl} Z & = & ([M]\mathsf{ff} \wedge [\varepsilon, gT, iTT]U \wedge lf \wedge r) \vee \\ & & ([M]\mathsf{ff} \wedge [\varepsilon, gT, iTT]U \wedge lf \wedge \neg r) \end{array}$$

The equations defining Y and X_{lf} are handled in a similar way. The only step that has some effect is step 5, which introduces the missing literal r. More interesting is how Phase I is applied to the new equation $U = Y \wedge X_{lf}$.

(1) Rewriting into strongly guarded form yields:

$$U = [M]\mathsf{ff} \wedge [\varepsilon, gT, iTT]Y \wedge [\varepsilon, lf, SI_P]X_{lf} \wedge lf$$

(2) Formula ϕ_U is already in DNF and cannot be simplified.
(3) Grouping boxes results in the following equation:

$$U = [M]\mathsf{ff} \wedge [\varepsilon, lf, SI_P](Y \wedge X_{lf}) \wedge lf$$

(4) The map h contains the pair $(\{Y, X_{lf}\}, U)$, so we replace $Y \wedge X_{lf}$ by U.

$$U = [M]\mathsf{ff} \wedge [\varepsilon, gT, iTT]U \wedge lf$$

(5) Literal completion again introduces r.

$$\begin{array}{lcl} U & = & ([M]\mathsf{ff} \wedge [\varepsilon, gT, iTT]U \wedge lf \wedge r) \vee \\ & & ([M]\mathsf{ff} \wedge [\varepsilon, gT, iTT]U \wedge lf \wedge \neg r) \end{array}$$

After applying Phase I to all equations, Phase II introduces a new equation for each disjunct and replaces each old variable by the disjunction of the new variables. For example, the equation defining U gets replaced by:

$$\begin{array}{lcl} U_1 & = & [M]\mathsf{ff} \wedge [\varepsilon, gT, iTT](U_1 \vee U_2) \wedge lf \wedge r \\ U_2 & = & [M]\mathsf{ff} \wedge [\varepsilon, gT, iTT](U_1 \vee U_2) \wedge lf \wedge \neg r \end{array}$$

The remaining equations are treated similarly. Notice that also Z in $\mathcal{X}$ gets replaced by $\{Z_1, Z_2\}$, where Z_1 and Z_2 are the equations replacing Z.

During the optimisation in Phase III, we find that the equations for Z_1 and U_1, and Z_2 and U_2 are duplicates of each other. Therefore, we remove the equations for Z_1 and Z_2, and replace $\{Z_1, Z_2\}$ in $\mathcal{X}$ by $\{U_1, U_2\}$. Further, the equations Y_1, Y_2, X_{lf1} and X_{lf2} (replacing Y and X_{lf} in Phase II), are not reachable from any variable in $\mathcal{X} = \{U_1, U_2\}$. Hence, the final result is $(U_1 \vee U_2)[\Sigma]$, where

$$\Sigma = \begin{bmatrix} U_1 = [M]\mathsf{ff} \wedge [\varepsilon, gT, iTT](U_1 \vee U_2) \wedge lf \wedge r \\ U_2 = [M]\mathsf{ff} \wedge [\varepsilon, gT, iTT](U_1 \vee U_2) \wedge lf \wedge \neg r \end{bmatrix}$$

The specification extracted from this modal equation system (which is in simulation normal form) is the maximal applet $\theta_{I_L}(\sigma_L)$ for σ_L. It is shown in Figure 7(a). The method graph has two nodes; both of them are entry points of the method, but only one is labelled as a return point. The edges are labelled only with internal actions and calls to *getTransaction* and *isThereTransaction*. As mentioned above, in the computation above we simplified SI_L to $\{lf\}$. If we do the computation for the complete shareable interface SI_L, we find that for all other methods m in SI_L, the method graph is a maximal method graph without restrictions, as in Figure 7(b). If we do the same computation for σ_P, we find the method graph for *isThereTransaction* in the maximal model for the *Purse* as in Figure 7(c), i.e., the method can only call itself or make internal transitions. The method graph for *getTransaction* is similar, with all edges labelled with *getTransaction* or ε, while the method graphs for the other methods provided by the *Purse* are maximal method graphs, without any restrictions.

Using our implementation of the maximal model construction in Ocaml, computing the maximal applets for σ_L and σ_P takes less than a second. Table 2 shows the relevant information.

Once the maximal applets $\theta_{I_P}(\sigma_P)$ and $\theta_{I_L}(\sigma_L)$ are constructed, we produce their composition $\theta_{I_P}(\sigma_P) \uplus \theta_{I_L}(\sigma_L)$, and we use the Model Generator to translate the applet graph to a PDA representation, serving as input to a PDA model checker.

Table 2
Size and timing for maximal applet construction

	$\theta_{I_L}(\sigma_L)$	$\theta_{I_P}(\sigma_P)$
# Nodes	8	8
# Edges	120	88
Constr. time (s)	0.05	0.05

Table 3
Statistics on applet graph extraction and verification

	# Classes	# Methods	# Nodes	# Edges	Extr. time (s)	Inline time (s)	Mod. gen. time (s)	Verif. time (s)
Loyalty	11	237	3 782	4 372	5.6	0.6	2.8	10.1
Purse	15	367	5 882	7 205	7.5	0.6	0.6	3.6

8.3.2. Correctness of the local structural properties

We use the Applet Analyser to extract applet graphs and the appropriate set of entry points from the byte code of the loyalty and purse implementations. Table 3 provides statistics on the extracted applet graphs.

Next, we applied the implementation of the inlining algorithm to the extracted applet graphs, which took 0.6 seconds on both *Loyalty* and *Purse*. Since the applets are last-call recursive, the inlining does not introduce any new observable interface behaviours. Even though theoretically the worst-case blowup in the number of nodes of the inlined applets, determined by the number of normal M-frames, is exponential in the number of private methods, in practice this is not likely to happen. In our case, we even observed a reduction in size of the graphs due to the following two facts: first, the call dependency graph is sparse and, second, the inlining focuses on interaction between applets, and thus any code that is not reachable by a shareable interface method is abstracted away by the inlining (as it is not relevant to the property we are interested in).

Finally, we used the Model Generator to translate the inlined applet graphs into input for CWB, and we verified the structural properties. Table 3 also provides statistics for the model generation and verification time.

Remark. Initially, we did not distinguish between public and private methods when we performed the case study (see [30]). This gave significant performance problems: the maximal applets contained implementations for (and calls to) all private methods as well, which resulted in huge structures. Moreover, without the distinction between public and private methods we had to compute the transitive closure of method calls to be able to express the local structural specifications, which resulted in a non-robust specification: for example splitting a private method into two would break the local specification. Adding the distinction between public and private methods thus resulted in a conceptually cleaner compositional verification method, with a drastically improved performance.

9. Conclusion

We have developed a compositional verification method for sequential programs with procedures. The method is particularly suited for supporting the secure dynamic loading of applets onto smart cards and other smart devices, but dynamically reconfiguring distributed systems based on remote procedure call communication also provides a suitable application area for the method. Using our verification method, secure dynamic loading can be achieved through the following scenario:

(1) Specify global security property ϕ (at structural or behavioural public level).
(2) For any initially unavailable applet $\mathcal{A}$ with public interface I containing public methods M, specify a local specification $\sigma_{\mathcal{A}}$ (at structural public level).
(3) Compute a maximal applet $\theta_I(\sigma_{\mathcal{A}})$, and verify that this maximal applet, composed with the inlining of the already available applets $\mathcal{B}$ (with public methods N) satisfies ϕ, i.e., verify $\theta_I(\sigma_{\mathcal{A}}) \uplus \alpha_N(\mathcal{B}) \models \phi$. This establishes the correctness of the decomposition.
(4) When applet $\mathcal{A}$ becomes available, compute its abstraction $\alpha_M(\mathcal{A})$ by inlining its private methods, and verify that this abstraction respects the local specification, i.e., $\alpha_M(\mathcal{A}) \models \sigma_{\mathcal{A}}$.

Notice that we restrict attention to control-flow safety properties. We have shown applicability of this approach on an industrial case study. To support the above scenario, we have developed the following theoretical contributions:

(1) a logical characterisation of simulation, and vice versa, a behavioural characterisation of logical satisfaction (for safety properties) in terms of maximal models;
(2) adaptation of the maximal model technique to procedural programs;
(3) a sound and complete compositional verification method for procedural programs; and
(4) a behaviour-preserving inlining transformation of procedural programs.

Future work The program model which forms the basis for our analyses is rather abstract. We are currently investigating how to extend our techniques to finer program models. In particular, we are considering program models capturing multi-

threading and exceptions. Our compositional verification principle remains valid, as long as the notions of structure and behaviour (and the corresponding notions of simulation and logic) can be extended so that the necessary technical conditions still apply. However, the verification problem for the global behavioural property becomes undecidable in the presence of multi-threading [45] (when considering the same primitives as in e.g., Java), thus appropriate abstraction techniques have to be employed for this task (as proposed in e.g., [46,47,48]). A further extension of significant interest is adding data to the program model, so that a more precise control flow can be modelled, and properties over data can be specified. This requires again the use of appropriate abstractions in order to retain decidability of the verification problems.

In principle, our verification technique can be extended to more expressive logics, for example to the full modal μ-calculus. However, adding diamond modalities and least fixed-point recursion to the logic requires a more general notion of model (and hence applet structures and behaviours) in the framework; for example, see [26,49] for such models and corresponding maximal model constructions.

Further, we are investigating under what restrictions one can construct maximal applets for behavioural properties, thus extending the method to deal with local behavioural properties. The approach we take is to define a translation from behavioural properties into collections of structural properties, such that any applet that is simulated by a maximal applet for one of the structural properties satisfies the original behavioural one. Preliminary results in this direction are presented in [11].

References

[1] Common Criteria. Available from: http://www.commoncriteriaportal.org.
[2] G. Chugunov, L. Fredlund, D. Gurov, Model checking of multi-applet JavaCard applications, Smart Card Research and Advanced Application Conference (CARDIS '02), USENIX Publications, 2002, pp. 87–95.
[3] J. Esparza, D. Hansel, P. Rossmanith, S. Schwoon, Efficient algorithms for model checking pushdown systems, Computer Aided Verification (CAV '00), LNCS, vol. 1855, Springer Verlag, 2000, pp. 232–247.
[4] F. Besson, T. Jensen, D. Le Métayer, T. Thorn, Model checking security properties of control flow graphs, Journal of Computer Security 9 (3) (2001) 217–250.
[5] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, M. Yannakakis, Analysis of recursive state machines, ACM TOPLAS 27 (2005) 786–818.
[6] O. Grumberg, D. Long, Model checking and modular verification, ACM TOPLAS 16 (3) (1994) 843–871.
[7] G. Barthe, D. Gurov, M. Huisman, Compositional verification of secure applet interactions, Fundamental Approaches to Software Engineering (FASE '02), LNCS, vol. 2306, Springer Verlag, 2002, pp. 15–32.
[8] D. Kozen, Results on the propositional μ-calculus, Theoretical Computer Science 27 (1983) 333–354.
[9] A. Bouajjani, J. Fernandez, S. Graf, C. Rodriguez, J. Sifakis, Safety for branching time semantics, Automata, Languages and Programming (ICALP '91), LNCS, vol. 501, Springer Verlag, 1991, pp. 76–92.
[10] O. Burkart, D. Caucal, F. Moller, B. Steffen, Verification on infinite structures, in: J. Bergstra, A. Ponse, S. Smolka (Eds.), Handbook of Process Algebra, North Holland, 2000, pp. 545–623.
[11] D. Gurov, M. Huisman, Reducing behavioural to structural properties of programs with procedures, Tech. Rep. TRITA-CSC-TCS 2007:3, KTH Royal Institute of Technology, Stockholm. 2007. Available from: http://www.csc.kth.se/~dilian/Papers/techrep-07-3.pdf.
[12] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, P. Co, Soot—a Java Optimization Framework, in: CASCON '99, 1999, pp. 125–135. Available from: http://www.sable.mcgill.ca/soot/.
[13] A. Lal, T.W. Reps, Improving pushdown system model checking, Computer Aided Verification (CAV '06), LNCS, vol. 4144, Springer Verlag, 2006, pp. 343–357.
[14] R. Alur, K. Etessami, P. Madhusudan, A temporal logic for nested calls and returns, Tools and Algorithms for the Analysis and Construction of Software (TACAS '04), LNCS, vol. 2998, Springer Verlag, 2004, pp. 467–481.
[15] R. Alur, M. Arenas, P. Barcelo, K. Etessami, N. Immerman, L. Libkin, First-order and temporal logics for nested words, Logic in Computer Science (LICS '07), IEEE Computer Society, Washington, DC, USA, 2007, pp. 151–160.
[16] W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, J. Zwiers, Concurrency Verification: Introduction to Compositional and Noncompositional Methods, No. 54 in Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 2001.
[17] K. Laster, O. Grumberg, Modular model checking of software, Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '98), LNCS, vol. 1384, Springer Verlag, 1998, pp. 20–35.
[18] R. Alur, R. Grosu, Modular refinement of hierarchic reactive machines, ACM TOPLAS 26 (2004) 339–360.
[19] O. Ly, Compositional verification: Decidability issues using graph substitutions, Proceedings of the 29th Mathematical Foundations of Computer Science (MFCS '04), LNCS, vol. 3153, Springer Verlag, 2004, pp. 537–549.
[20] H. Andersen, Partial model checking (extended abstract), Logic in Computer Science (LICS '95), IEEE Computer Society Press, 1995, pp. 398–407.
[21] O. Kupferman, M. Vardi, An automata-theoretic approach to modular model checking, ACM TOPLAS 22 (1) (2000) 87–128.
[22] G. Boudol, K. Larsen, Graphical versus logical specifications, Theoretical Computer Science 106 (1992) 3–20.
[23] K. Larsen, Modal specifications, Automatic Verification Methods for Finite State Systems, LNCS, vol. 407, Springer Verlag, 1989, pp. 232–246.
[24] M. Hennessy, R. Milner, Algebraic laws for nondeterminism and concurrency, Journal of the ACM 32 (1985) 137–161.
[25] D. Dams, K. Namjoshi, The existence of finite abstractions for branching time model checking, Nineteenth Annual IEEE Symposium on Logic in Computer Science (LICS '04), IEEE Computer Society Press, 2004, pp. 335–344.
[26] D. Dams, K. Namjoshi, Automata as abstractions, Verification, Model Checking, and Abstract Interpretation (VMCAI '05), LNCS, vol. 3385, Springer Verlag, 2005, pp. 216–232.
[27] M. Goldman, S. Katz, MAVEN: Modular aspect verification, Tools and Algorithms for the Construction and Analysis of Systems (TACAS '07), LNCS, vol. 4424, Springer Verlag, 2007, pp. 308–322.
[28] C. Sprenger, D. Gurov, M. Huisman, Compositional verification for secure loading of smart card applets, Formal Methods and Models for Co-Design (MEMOCODE '04), IEEE Computer Society, 2004, pp. 211–222.
[29] D. Gurov, M. Huisman, Interface abstraction for compositional verification, Software Engineering and Formal Methods (SEFM'05), IEEE Computer Society, 2005, pp. 414–423.
[30] M. Huisman, D. Gurov, C. Sprenger, G. Chugunov, Checking absence of illicit applet interactions: a case study, Fundamental Approaches to Software Engineering (FASE '04), LNCS, vol. 2984, Springer Verlag, 2004, pp. 84–98.
[31] H. Bekič, Definable operators in general algebras, and the theory of automata and flowcharts, Tech. Rep., IBM Laboratory, 1967.
[32] C. Stirling, Modal and Temporal Logics of Processes, Springer Verlag, 2001.
[33] A. Tarski, A lattice-theoretical fixpoint theorem and its applications, Pacific Journal of Mathematics 5 (1955) 285–310.
[34] A. Arnold, D. Niwiński, Rudiments of μ-calculus, Studies in Logic and the Foundations of Mathematics, vol. 146, Elsevier Publishing, 2001.

[35] I. Walukiewicz, Pushdown processes: games and model checking, in: Computer Aided Verification (CAV '96), LNCS, vol. 1102, 1996, pp. 62–75.
[36] C. Sprenger, D. Gurov, M. Huisman, Simulation logic, applets and compositional verification, Tech. Rep. RR-4890, INRIA, 2003.
[37] D. Gurov, M. Huisman, Abstraction over public interfaces, Tech. Rep. RR-5330, INRIA, 2004.
[38] R. Cleaveland, J. Parrow, B. Steffen, A semantics based verification tool for finite state systems, International Symposium on Protocol Specification, Testing and Verification, North-Holland Publishing Co., Amsterdam, The Netherlands, 1990, pp. 287–302.
[39] D. Polanský, Verifying properties of infinite-state systems, Master's thesis, Masaryk University, Faculty of Informatics, Brno, 2000.
[40] A. Bouajjani, J. Esparza, O. Maler, Reachability analysis of pushdown automata: Application to model-checking, in: International Conference on Concurrency Theory (CONCUR '97), vol. 1243 of LNCS, 1997, pp. 135–150.
[41] E. Bretagne, A.E. Marouani, P. Girard, J.-L. Lanet, PACAP purse and loyalty specification, Tech. Rep. V 0.4, Gemplus, 2000.
[42] C. Breunesse, N. Cataño, M. Huisman, B. Jacobs, Formal methods for smart cards: an experience report, Science of Computer Programming 55 (1–3) (2005) 53–80.
[43] P. Bieber, J. Cazin, P. Girard, J.-L. Lanet, V. Wiels, G. Zanon, Checking secure interactions of smart card applets, Journal of Computer Security 10 (4) (2002) 369–398.
[44] J. Corbett, M. Dwyer, J. Hatcliff, Robby, A language framework for expressing checkable properties of dynamic software, International SPIN Workshop on SPIN Model Checking and Software Verification, LNCS, vol. 1885, Springer Verlag, 2000, pp. 205–223.
[45] G. Ramalingam, Context-sensitive synchronization-sensitive analysis is undecidable, ACM TOPLAS 22 (2) (2000) 416–430.
[46] A. Bouajjani, J. Esparza, T. Touili, A generic approach to the static analysis of concurrent programs with procedures, SIGPLAN Notes 38 (1) (2003) 62–73.
[47] A. Bouajjani, J. Esparza, S. Schwoon, J. Strejček, Reachability analysis of multithreaded software with asynchronous communication, Foundations of Software Technology and Theoretical Computer Science (FSTTCS '05), LNCS, vol. 3821, Springer Verlag, 2005, pp. 348–359.
[48] S. Qadeer, J. Rehof, Context-bounded model checking of concurrent software, Tools and Algorithms for the Construction and Analysis of Systems (TACAS '05), LNCS, vol. 3440, Springer Verlag, 2005, pp. 93–107.
[49] I. Aktug, D. Gurov, State space representation for verification of open systems, Algebraic Methodology And Software Technology (AMAST '06), LNCS, vol. 4019, Springer Verlag, 2006, pp. 5–20.