Reducing Behavioural to Structural Properties of Programs with Procedures

Dilian Gurov[1,⋆] and Marieke Huisman[2,⋆⋆]

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

Abstract. There is an intimate link between program structure and behaviour. Exploiting this link to phrase program correctness problems in terms of the structural properties of a program graph rather than in terms of its unfoldings is a useful strategy for making analyses more tractable. This paper presents a characterisation of behavioural program properties through sets of structural properties by means of a translation. The characterisation is given in the context of a program model based on control flow graphs of sequential programs with possibly recursive procedures, and properties expressed in a fragment of the modal μ-calculus with boxes and greatest fixed-points only. The property translation is based on a tableau construction that conceptually amounts to symbolic execution of the behavioural formula, collecting structural constraints along the way. By keeping track of the subformulae that have been examined, recursion in the structural constraints can be identified and captured by fixed-point formulae. The tableau construction terminates, and the characterisation is exact, *i.e.,* the translation is sound and complete. A prototype implementation has been developed. We discuss several applications of the characterisation, in particular compositional verification for behavioural properties, based on maximal models.

1 Introduction

The relationship between a program's syntactical structure and its behaviour is fundamental in program analysis. For example, type systems analyse the structure of a program to deduce properties about its behaviour, while program synthesis studies how to realise a program structure for a desired program behaviour. The relationship is often exploited to phrase program correctness problems in terms of the structure of a program rather than in terms of its behaviour, in order to make analyses more tractable. If program data is abstracted away, and only the *control flow* of programs with (possibly recursive) procedures is considered, the relation between structure and behaviour is well-understood in one

⋆ Partially funded by the IST FP6 programme of the EC, under the IST-FP6-STREP-27004 S3MS project.
⋆⋆ Work done while at INRIA Sophia Antipolis. Partially funded by the IST FET programme of the EC, under the IST-2005-015905 MOBIUS project.

N.D. Jones, M. Müller-Olm (Eds.): VMCAI 2009, LNCS 5403, pp. 136–150, 2009.

direction: program structure, essentially a finite "program graph", can be represented by a pushdown system that induces program behaviour as an "unfolding" of the structure in a context-free manner. This representation has been exploited widely, for example for interprocedural dataflow analysis (*e.g.,* in [17]) and for model checking of behavioural properties (*e.g.,* in [8]). However, in the other direction, this relationship is much less understood: given a program behaviour, how can one capture the program structures that admit this behaviour?

Both program structure and behaviour can be specified by *temporal logic* formulae: structural properties are concerned with the textual sequencing of instructions in a program, while behavioural properties consider their executional sequencing. The relationship between structure and behaviour is naturally expressed at the logic level through the following two questions:

(1) when does a structural property entail a behavioural one and,
(2) can a behavioural property be characterised by a finite set of structural ones?

This extended abstract (the accompanying report [11] contains proofs and more examples) addresses this *characterisation problem* in the context of a program model based on control flow graphs of sequential programs with procedures (*i.e.*, program data is abstracted away), for properties expressed in a fragment of the modal μ-calculus with boxes and greatest fixed-points only. This temporal logic is suitable for expressing safety properties (*cf.* [4]) in terms of sequences of method invocations, such as security policies restricting access to given resources by means of API method calls (*cf.* [18]). In previous work [12], we showed how this logic can be used for the specification and compositional verification of safety properties, both on the structural and on the behavioural level, and provided tool support and case studies. In particular, we derived an algorithmic solution to problem (1) stated above (see [12, p. 855]). Here, we give a precise solution to the (more complex) problem (2), showing that every disjunction-free behavioural formula can be characterised by a finite set of structural formulae: a program satisfies the behavioural formula if and only if it satisfies *some* structural formula from the set. For example, the results of this paper allow to derive that the behavioural property "method a never calls method b" is characterised by the (singleton set) structural property "in (the text of) method a, every call–to–b instruction is preceded by some call–to–a instruction" (and hence, due to recursion, control never reaches a call–to–b instruction).

Our solution is constructive, by means of a translation Π from behavioural properties into sets of structural properties. The translation has been implemented in Ocaml and can be tested online [10]. It conceptually amounts to a symbolic execution of the behavioural formula, collecting induced structural constraints along the way. A considerable difficulty is presented by (greatest fixed-point) recursion in the behavioural formula, which has to be captured by recursion in the structural ones (in the absence of recursion it is considerably easier to define such a translation, as we show in [14]). We handle recursion by means of a *tableau construction* that maintains (during the symbolic execution) a symbolic "call stack" indicating which subformulae have been explored for which method. We use this stack to (1) identify when a (sub)formula has

been sufficiently explored, so that a branch of the tableau can be finished, and (2) to identify recursion in the collected structural constraints and capture this by fixed-point formulae. We prove that the construction terminates. Moreover, we show that the construction is *sound*, and in case the behavioural formula is disjunction-free, also *complete,* by viewing the tableau system as a proof system.

Applications. In addition to its foundational value, the characterisation is useful in various ways. In earlier work, we defined a *maximal model* construction for the logic considered here, and adapted it to the construction of maximal program structures from structural properties [12]. The combination of this construction with the property translation Π provides a solution to the problem of computing maximal program structures from behavioural properties. As Section 4 shows, this can be exploited to extend the *compositional verification* technique of [12], where local assumptions are required to be structural, to local behavioural properties. Further, the translation can be used to reduce infinite-state verification of behavioural control flow properties to finite-state verification of structural properties. Thus, tools supporting structural properties only can in effect be used for verifying behavioural properties. Moreover, in a mobile code deployment scheme, where the security policies of the platform are given as behavioural control flow properties, translating these into structural properties of the loaded applications enables efficient on-device conformance checking via static analysis.

Related Work. Our property translation has been motivated by our previous work on adapting Grumberg and Long's approach of using maximal models for compositional verification [9] in the context of control flow properties of sequential programs with procedures [12]. Maximal models can also be constructed for the full μ-calculus, but require representations beyond ordinary labelled transition systems, such as the focused transition systems proposed by Dams and Namjoshi [7]. Our research is also related to previous work on tableau systems for the verification of infinite-state systems [6,19], model checking based on pushdown systems [5,8] or recursive state machines [2], temporal logics for nested calls and returns [1,3], interprocedural dataflow analysis [17], and abstract interpretation (*cf. e.g.,* the completeness result of [16]). However, these analyses infer from the structure of a given program facts about its behaviour; in contrast, our analysis infers, for *all* programs satisfying a certain behaviour, facts about the structure from facts about that behaviour.

Organisation. Section 2 formally defines the program model and logic. Next, Section 3 defines the translation, by means of the tableau construction. Section 4 uses the characterisation to develop a sound and complete compositional verification principle for local behavioural properties, while Section 5 concludes with a discussion of possible extensions and optimisations.

2 Preliminaries: Program Model and Logic

This section summarises the definitions of program model, logic and satisfaction. These are first defined generally, and then instantiated at structural and behavioural level. We refer the reader to [12] for more details.

2.1 Specification and Logic

First, we define the general notions of model and specification.

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

As property specification language, we use the fragment of the modal μ-calculus [15] with boxes and greatest fixed-points only. This fragment is suitable for expressing safety properties and is capable of characterising simulation (*cf.* [12]). Throughout, we fix a set of labels L, a set of atomic propositions A, and a set of propositional variables V.

Definition 2. (Logic) *The formulae of our logic are inductively defined by:* $\phi ::= p \mid \neg p \mid X \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid [a]\,\phi \mid \nu X.\phi$*, where* $p \in A$*,* $a \in L$ *and* $X \in V$*.*

Satisfaction on states $(\mathcal{M}, s) \models \phi$ (also denoted $s \models^{\mathcal{M}} \phi$) is defined in the standard fashion [15]. For instance, formula $[a]\,\phi$ holds of state s in model $\mathcal{M}$ if ϕ holds in all states accessible from s via a transition labelled a. A specification $(\mathcal{M}, E)$ satisfies a formula if all its entry states E satisfy the formula. The constant formulae *true* (denoted tt) and *false* (ff) are definable. For convenience, we use $p \Rightarrow \phi$ to abbreviate $\neg p \vee \phi$. In our translation of simulation logic formulae we allow sequences α of labels to appear in box modalities, with the obvious translation $\widehat{\ .\ }$ to standard formulae: $\widehat{[\epsilon]\,\psi} = \psi$ and $\widehat{[l \cdot \alpha]\,\psi} = [l]\,\widehat{[\alpha]\,\psi}$, where ϵ denotes the empty sequence, and ψ is already a standard formula.

2.2 Control Flow Structure and Behaviour

Our program model is control–flow based and thus over–approximates actual program behaviour. This approach is sound, since we focus on safety properties. We define two different views on programs: a structural and a behavioural view. Both views are instantiations of the general notions of model and specification. Notice in particular that these instantiations yield a structural and a behavioural version of the logic.

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

Definition 3. (Method specification) *A* flow graph *for* $m \in Meth$ *over a set* $M \subseteq Meth$ *of method names is a finite model* $\mathcal{M}_m = (V_m, L_m, \rightarrow_m, A_m, \lambda_m)$*, with* V_m *the set of* control nodes *of* m*,* $L_m = M \cup \{\varepsilon\}$*,* $A_m = \{m, r\}$*, and* $\lambda_m\colon V_m \rightarrow \mathcal{P}(A_m)$ *so that* $m \in \lambda_m(v)$ *for all* $v \in V_m$ *(*i.e.*, each node is tagged*

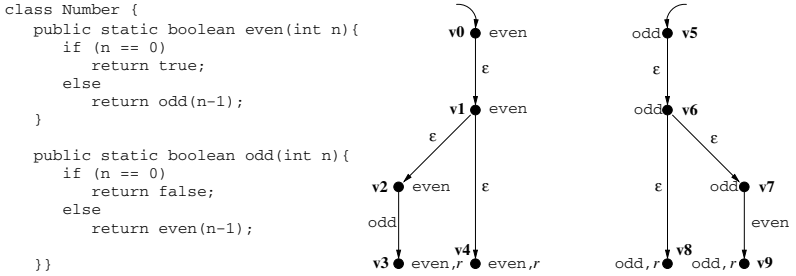

Fig. 1. A simple Java class and its flow graph

with its method name). The nodes $v \in V_m$ *with* $r \in \lambda_m(v)$ *are* return points. *A* method specification *for* $m \in Meth$ *over* M *is a pair* $(\mathcal{M}_m, E_m)$, *s.t.* $\mathcal{M}_m$ *is a flow graph for* m *over* M *and* $E_m \subseteq V_m$ *a non–empty set of* entry points *of* m.

Next, we define flow graph interfaces. These ensure that control flow graphs can only be composed if their interfaces match.

Definition 4. (Flow graph interface) *A* flow graph interface *is a pair* $I = (I^+, I^-)$, *where* $I^+, I^- \subseteq Meth$ *are finite sets of names of* provided *and* required *methods, respectively. The* composition *of two interfaces* $I_1 = (I_1^+, I_1^-)$ *and* $I_2 = (I_2^+, I_2^-)$ *is defined by* $I_1 \cup I_2 = (I_1^+ \cup I_2^+, I_1^- \cup I_2^-)$. *An interface* $I = (I^+, I^-)$ *is* closed *if* $I^- \subseteq I^+$.

The flow graph of a program is essentially the (disjoint) union of its method graphs. To formally define the notion *flow graph with interface*, we use the notion of disjoint union of specifications $\mathcal{S}_1 \uplus \mathcal{S}_2$, where each state is tagged with 1 or 2, respectively, and $(s, i) \xrightarrow{a}_{\mathcal{S}_1 \uplus \mathcal{S}_2} (t, i)$, for $i \in \{1, 2\}$, if and only if $s \xrightarrow{a}_{\mathcal{S}_i} t$.

Definition 5. (Flow graph with interface) *A* flow graph $\mathcal{G}$ with interface I, *written* $\mathcal{G} : I$, *is defined inductively by*

- $(\mathcal{M}_m, E_m) : (\{m\}, M)$ *if* $(\mathcal{M}_m, E_m)$ *is a method specification for* $m \in Meth$ *over* M, *and*
- $\mathcal{G}_1 \uplus \mathcal{G}_2 : I_1 \cup I_2$ *if* $\mathcal{G}_1 : I_1$ *and* $\mathcal{G}_2 : I_2$.

A flow graph is closed if its interface is closed (*i.e.,* it does not require any external methods), and is *clean* if return points have no outgoing edges. In the sequel, we shall assume, without loss of generality, that flow graphs are clean. Satisfaction, instantiated to flow graphs, is called structural satisfaction $\models_s$.

Example 1. Figure 1 shows a Java class and its (simplified) flow graph with interface $(\{\texttt{even}, \texttt{odd}\}, \{\texttt{even}, \texttt{odd}\})$. This contains two method specifications, for method `even` and for method `odd`, respectively. Entry nodes are depicted as usual by incoming edges without source. For this flow graph, the structural formula $\nu X.\, [\texttt{even}]\, r \wedge [\texttt{odd}]\, r \wedge [\varepsilon]\, X$ expresses the property that "on every path from a program entry node, the first encountered call edge leads to a return node", in effect specifying that the program is tail-recursive.

Control Flow Behaviour. Next, we instantiate specifications on the behavioural level. We use transition label τ to designate internal transfer of control, label $m_1\ \mathsf{call}\ m_2$ to designate an invocation of method m_2 by method m_1, and label $m_2\ \mathsf{ret}\ m_1$ for a corresponding return from the call.

Definition 6. (Behaviour) *Let $\mathcal{G} = (\mathcal{M}, E) : I$ be a closed flow graph where $\mathcal{M} = (V, L, \rightarrow, A, \lambda)$. The* behaviour *of $\mathcal{G}$ is defined as model $b(\mathcal{G}) = (\mathcal{M}_b, E_b)$, where $\mathcal{M}_b = (S_b, L_b, \rightarrow_b, A_b, \lambda_b)$, such that $S_b = V \times V^*$,* i.e., *states (or* configurations*) are pairs of control points v and stacks σ, $L_b = \{m_1\ k\ m_2 \mid k \in \{\mathsf{call}, \mathsf{ret}\}, m_1, m_2 \in I^+\} \cup \{\tau\}$, $A_b = A$, $\lambda_b((v,\sigma)) = \lambda(v)$, and $\rightarrow_b \subseteq S_b \times L_b \times S_b$ is defined by the rules:*

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

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

Flow graph behaviour can alternatively be defined in terms of *pushdown automata* (PDA) [12, Def. 34]. This can be exploited by using PDA model checking for verifying behavioural properties (see for instance [5,8]).

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

Example 2. Consider the flow graph from Example 1. Because of possible unbounded recursion, it induces an infinite-state behaviour. One example run (*i.e.*, linear execution) through this behaviour is represented by the following path from an initial to a final configuration:

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

For this flow graph, the behavioural formula $\mathtt{even} \Rightarrow \nu X.\ [\mathtt{even}\ \mathsf{call}\ \mathtt{even}]\,\mathsf{ff} \wedge [\tau]\,X$ expresses the property "in every program execution that starts in method `even`, the first call is not to method `even` itself".

More example properties and realistic program specifications can be found in [12].

3 Mapping Behavioural into Structural Properties

This section defines a mapping Π from interfaces and behavioural properties to sets of structural properties. As mentioned above, the implementation of the mapping can be tested online. Throughout the section we assume that behavioural properties are disjunction-free; in Section 5 we discuss how Π can be

extended to behavioural formulae with disjunction, though at the expense of completeness. We show that Π computes, from a behavioural property ϕ and closed interface I, a set of structural formulae that characterises ϕ and I. That is, for any (closed) flow graph $\mathcal{G}$ with interface I and any behavioural formula ϕ that only mentions labels that are in the behaviour of $\mathcal{G}$ (*i.e.*, L_b in Definition 6):

$$\mathcal{G} \models_b \phi \;\Leftrightarrow\; \exists \chi \in \Pi_I(\phi).\, \mathcal{G} \models_s \chi \tag{1}$$

To deal with the fixed-point formulae of the logic, mapping Π is defined with the help of a *tableau construction*. A behavioural formula ϕ gives rise to a (maximal) tableau that induces a set of structural formulae through its leaves. The constructed tableau is finite, *i.e.,* tableau construction terminates.

3.1 Tableau Construction

Our translation is based on a *symbolic execution* of the behavioural property by means of a *tableau construction*. When tracing a symbolic execution path, we tag all subformulae of the formula with unique propositional constants from a set $\mathcal{C}onst$. We use a global map $\mathcal{S} : \phi \to \mathcal{C}onst$ to map formulae to their tags. We consider $\mathcal{S}$ as an implicit parameter of the tableau construction. The tableau construction operates on sequents of the shape $\vdash_{H,U,C} \phi$ parametrised on:

- a non-empty *history stack* $H \in (I^+ \times (I^- \cup \{\varepsilon\} \cup \mathcal{C}onst)^*)^+$, where each element is a pair (i, F) consisting of a method name $i \in I^+$ (called the current method) and a sequence $F \in (I^- \cup \{\varepsilon\} \cup \mathcal{C}onst)^*$ of edge labels and propositional constants abbreviating subformulae of ϕ (called frame). For any frame F, we use $\widetilde{F}$ to denote this frame cleaned from propositional constants $X \in \mathcal{C}onst$:

$$\widetilde{\epsilon} = \epsilon \qquad \widetilde{m \cdot \sigma} = m \cdot \widetilde{\sigma} \qquad \widetilde{\varepsilon \cdot \sigma} = \varepsilon \cdot \widetilde{\sigma} \qquad \widetilde{X \cdot \sigma} = \widetilde{\sigma}$$

- a *fixed-point stack* U, defining an environment for propositional variables by means of a sequence of definitions of the shape $X = \nu X.\psi$. An *open* formula ϕ in a sequent parametrised by U can then be understood via a suitable notion of substitution, based on the standard notion of substitution $\psi\{\theta/X\}$ of a formula θ for a propositional variable X in a formula ψ: the *substitution* of ϕ under U is inductively defined as follows:

$$\phi[\epsilon] = \phi \qquad \phi[(X = \nu X.\psi) \cdot U] = (\phi\{\nu X.\psi/X\})[U]$$

- a *store* C, used for accumulating structural constraints during symbolic execution.

We use $\varnothing_{H,m}$, $\varnothing_U$ and $\varnothing_C$ to denote the single-element history stack (m, ϵ) and the empty fixed-point stack and store, respectively.

For a given closed behavioural formula ϕ and method m, we construct a maximal tableau with root $\vdash_{\varnothing_{H,m},\varnothing_U,\varnothing_C} \phi$ that induces a set of structural formulae through its leaves, as described below. We denote the set of induced structural

formulae for ϕ and m with $\pi_m(\phi)$. We then define the translation of ϕ *w.r.t.* a given interface I:

$$\Pi_I(\phi) = \{ \bigwedge_{m \in I^+} \chi_m \mid \chi_m \in \pi_m(\phi) \}$$

During tableau construction, the history stack, fixed-point stack and store are updated as follows, provided the current sequent is not a repeat of an earlier sequent (see below):

1. First, if ϕ is not a fixed-point formula, the propositional constant $\mathcal{S}(\phi)$ tagging the behavioural property ϕ of the current sequent is appended to the end of the frame of the top element of H;
2. Next,
   - if the behavioural property ϕ prescribes an internal transfer, then ε is appended to the end of the frame of the top element of H;
   - if ϕ prescribes a call from a to b, and the top element of H is in method a, then b is added at the end of the frame of the top element of H, and a new element (b, ϵ) is pushed onto H;
   - if ϕ prescribes a return from a to b, the top element of H is in method a and the next element is in method b, then a new structural constraint is added to the store, reflecting the possibility of currently not being at a return point, and the top element is popped from H; and
   - if ϕ is a fixed-point formula $\nu X.\phi$, then a new equation $X = \nu X.\phi$ is pushed onto the fixed-point stack U, if not already there; this conditional appending is denoted by $(X = \nu X.\phi) \circ U$.

Notice that non-emptiness of the history stack and closedness of $\phi[U]$ are invariants of the tableau construction.

Tableau System. The tableau system is given in Figure 2 as a set of goal-directed rules. Axioms are presented as rules with an empty set of premises, denoted by '$-$'. The condition $\mathsf{Ret}(i, a, b, H)$ used in the return rules is defined as $i = a \wedge H \neq \epsilon \wedge \exists F, H'.H = (b, F) \cdot H'$, *i.e.,* control is currently in method a, the call stack is not empty, and the control point on the top of the stack is in method b. Formally, a *tableau* $\mathcal{T} = (T, \lambda)$ is a tree T equipped with a labelling function λ mapping each tree node to a triple consisting of a sequent, a rule name (of the rule applied to this sequent), and a set of triples of shape (i, F, q) where q are literals (that is, atomic propositions in positive or negated form or propositional variables). The triple sets are non-empty only at applications of axiom rules; such leaves are termed *contributing*, and the set of triples is depicted (by convention) as a premise to the rule. A tableau for formula ϕ and method m is a tree with root $\vdash_{\varnothing_H, m, \varnothing_U, \varnothing_C} \phi$ obtained by applying the rules. A tableau is termed *maximal* if all its leaves are axioms.

If in a tableau there is a leaf node $\vdash_{(i,F)\cdot H, U, C} \phi$ for which there is an internal node $\vdash_{(i,F')\cdot H', U', C'} \phi$ such that F' is a prefix of F, U' is a suffix of U, and C' is a subset of C, we term the former node a *pseudo-repeat*; any node of the latter kind we term a *companion*. An internal tableau node is said to be *stable* if all its

$$p\ \frac{\vdash_{(i,F)\cdot H,U,C} p}{\{(i,F,p)\}\cup\{(i',F',\mathsf{ff})\mid(i',F')\in H\}\cup C} \qquad \neg p\ \frac{\vdash_{(i,F)\cdot H,U} \neg p}{\{(i,F,\neg p)\}\cup\{(i',F',\mathsf{ff})\mid(i',F')\in H\}\cup C}$$

$$\nu X\ \frac{\vdash_{(i,F)\cdot H,U,C} \nu X.\phi}{\vdash_{(i,F)\cdot H,(X=\nu X.\phi)\circ U,C} X} \qquad X\ \mathrm{unf}\ \frac{\vdash_{(i,F)\cdot H,U,C} X}{\vdash_{(i,F\cdot\mathcal{S}(X))\cdot H,U,C} \phi}\ (X=\nu X.\phi)\in U$$

$$\wedge\ \frac{\vdash_{(i,F)\cdot H,U,C} \phi_1\wedge\phi_2}{\vdash_{(i,F\cdot\mathcal{S}(\phi_1\wedge\phi_2))\cdot H,U,C} \phi_1 \qquad \vdash_{(i,F\cdot\mathcal{S}(\phi_1\wedge\phi_2))\cdot H,U,C} \phi_2} \qquad \tau\ \frac{\vdash_{(i,F)\cdot H,U,C} [\tau]\phi}{\vdash_{(i,F\cdot\mathcal{S}([\tau]\phi)\cdot\varepsilon)\cdot H,U,C} \phi}$$

$$\mathrm{call}_0\ \frac{\vdash_{(i,F)\cdot H,U,C} [a\ \mathsf{call}\ b]\phi}{-}\ i\neq a \qquad \mathrm{call}_1\ \frac{\vdash_{(i,F)\cdot H,U,C} [a\ \mathsf{call}\ b]\phi}{\vdash_{(b,\epsilon)\cdot(i,F\cdot\mathcal{S}([a\ \mathsf{call}\ b]\phi)\cdot b)\cdot H,U,C} \phi}\ i=a$$

$$\mathrm{ret}_0\ \frac{\vdash_{(i,F)\cdot H,U,C} [a\ \mathsf{ret}\ b]\phi}{-}\ \neg\mathsf{Ret}(i,a,b,H) \qquad \mathrm{ret}_1\ \frac{\vdash_{(i,F)\cdot H,U,C} [a\ \mathsf{ret}\ b]\phi}{\vdash_{H,U,C\cup\{(i,F,\neg r)\}} \phi}\ \mathsf{Ret}(i,a,b,H)$$

$$\mathrm{IRep}\ \frac{\vdash_{(i,F)\cdot H,U,C} \phi}{\{(i,F,\mathcal{S}(\phi))\}\cup C}\ \mathsf{IntRep}(\mathcal{S}(\phi),(i,F)\cdot H) \qquad \mathrm{CRep}\ \frac{\vdash_{(i,F)\cdot H,U,C} \phi}{-}\ \mathsf{CallRep}(\mathcal{S}(\phi),(i,F)\cdot H,c)$$

$$\mathrm{RRep}\ \frac{\vdash_{(i,F)\cdot H,U,C} \phi}{-}\ \mathsf{RetRep}(\mathcal{S}(\phi),(i,F)\cdot H,c)$$

Fig. 2. Tableau system

descendant leaves are axioms or pseudo-repeats. A tableau is stable if its root node is stable.

Tableau construction proceeds as follows. First, a minimal stable tableau is computed, *i.e.,* if a node is a pseudo-repeat, it is not further explored. If all pseudo-repeats in this tableau satisfy some repeat condition for any of their companions (see below), the tableau is maximal and construction is complete. Otherwise, all pseudo-repeats that are not satisfying any of the repeat conditions are simultaneously unfolded, using a breadth-first exploration strategy, and tableau construction continues until the tableau is stable again, upon which the checking for the repeat conditions is repeated. This process is guaranteed to terminate, as we state later, resulting in a finite maximal tableau.

Repeat Conditions. We now formulate the three repeat conditions used in the tableau system, giving rise to three types of repeat nodes. Only repeats of the first type, *i.e.,* internal repeats, contribute to triples, giving rise to recursion in structural formulae. In contrast, the other two repeat conditions only recognise that a similar situation has been reached before, and thus no new information will be obtained by further exploration. The first repeat condition requires merely the examination of the top frame of the history stack of the current sequent; the second requires the examination of the whole path from the root to the pseudo-repeat; while the third requires the examination of all remaining paths.

Internal repeat. Tableau construction guarantees that every tableau node of shape $\vdash_{H'\cdot(i,F'\cdot\mathcal{S}(\phi)\cdot F'')\cdot H'',U,C}\ \phi$ possesses an ancestor node $\vdash_{(i,F')\cdot H'',U',C'}\ \phi$ such that U' is a suffix of U and C' is a subset of C. As a consequence, every node of shape $\vdash_{(i,F'\cdot\mathcal{S}(\phi)\cdot F'')\cdot H,U,C}\ \phi$ is a pseudo-repeat (with some ancestor node of shape $\vdash_{(i,F')\cdot H,U',C'}\ \phi$ as companion); such pseudo-repeats are termed *internal repeats*. Intuitively, an internal repeat indicates that a regularity in the

structure of method i has been discovered, and thus this regularity should be reflected in the structural formulae. Therefore, in this case $(i, F' \cdot \mathcal{S}(\phi) \cdot F'', \mathcal{S}(\phi))$ is added to the triple set of the IRep axiom. (Notice that in fact the propositional constant $\mathcal{S}(\phi)$ is mapped to a fresh propositional variable, here, and in the construction of the structural formulae. However, for clarity of presentation, we overload the symbols themselves, as their intended meaning should be clear from the context.)

Call repeat. A pseudo-repeat $\vdash_{(i,F)\cdot H,U,C} \phi$, which has an ancestor node as companion but is not an internal repeat, is a *call repeat* if H matches the call stack of the companion upto the latter's *return depth* (where matching means that the same methods are on the stack, with identical frames); in the special case where both stacks are shorter than the return depth, they have to be identical.

The return depth of a node is only defined if the subtableau of the companion is complete (*i.e.,* the pseudo-repeat is the only open branch). When we construct a tableau for a formula with multiple fixed-points, it can happen that two pseudo-repeats occur in the subtableaux of their respective companions. In this case, if both nodes are call repeats exploration terminates (for the current return depth); otherwise, by virtue of the tableau construction, the pseudo-repeat that is not a call repeat will never become one when continuing the tableau construction. Therefore, we can explore this node further, and break the mutual dependency.

The return depth of a tableau node n, denoted $\rho(n)$, is defined as the maximal difference between the number of applied return rules and the number of applied call rules on any path from n to a descendant node. Formally, where r and δ range over rule names and sequences of rule names, respectively, while $rules(\pi)$ denotes the sequence of rule names along a tableau path π:

$$\rho'(\epsilon) = 0 \qquad \rho'(r \cdot \delta) = \begin{cases} \rho'(\delta) + 1 & \text{if } r \in \{\mathsf{ret}_0, \mathsf{ret}_1\} \\ \rho'(\delta) - 1 & \text{if } r \in \{\mathsf{call}_0, \mathsf{call}_1\} \\ \rho'(\delta) & \text{otherwise} \end{cases}$$

$$\rho(n) = max\,\{\rho'(rules(\pi)) \mid \pi \text{ a path from } n \text{ to a descendant node}\} \cup \{0\}$$

Return repeat. A pseudo-repeat is called a *return repeat* if it has a companion on a different path from the root, such that its history stack is identical to the one of the companion.

Formally, the repeat conditions are defined as follows, where X is $\mathcal{S}(\phi)$, and c is the companion node of the pseudo-repeat with history stack H_c.

$$\begin{aligned} \mathsf{IntRep}(X, (i,F) \cdot H) &\Leftrightarrow X \in F \\ \mathsf{CallRep}(X, (i,F) \cdot H, c) &\Leftrightarrow X \notin F \wedge \mathsf{take}(\rho(c)+1, (i,F) \cdot H) = \mathsf{take}(\rho(c)+1, H_c) \\ \mathsf{RetRep}(X, (i,F) \cdot H, c) &\Leftrightarrow (i,F) \cdot H = H_c \end{aligned}$$

Termination. The repeat conditions ensure termination of tableau construction.

Theorem 1. *Maximal tableaux are finite.*

The proof of this and the remaining results can be found in [11].

3.2 Structural Formulae Induced by a Tableau

A maximal tableau for ϕ and m induces, through the sets of triples accumulated in the leaves, a set of structural formulae $\pi_m(\phi)$ in the following manner:

1. Let $\mathcal{L}$ be the set of non-empty triple sets collected from the leaves of the tableau. Build a collection of *choice sets* $\Lambda(\mathcal{L})$, by choosing one triple from each element in $\mathcal{L}$.
2. For each choice set $\lambda \in \Lambda(\mathcal{L})$,
   (a) Group the triples of λ according to method names: for each $i \in I$, define
   $$\Xi_i = \{(F, q) \mid (i, F, q) \in \lambda\}$$
   (b) For each $i \in I$ such that $\Xi_i \neq \varnothing$, build a formula $i \Rightarrow \Omega(\Xi_i)$, where
   $$\begin{array}{rl} \Omega(\Xi) = & \bigwedge_{\phi \in \Omega'(\Xi)} \phi \\ \Omega'(\Xi) = & \{[a]\,\Omega(\Xi') \mid a \in I^- \wedge \Xi' = \{(F, q) \mid (a \cdot F, q) \in \Xi\} \wedge \Xi' \neq \varnothing\} \cup \\ & \{\nu X.\Omega(\Xi') \mid X \in \mathcal{C}onst \wedge \Xi' = \{(F, q) \mid (X \cdot F, q) \in \Xi\} \wedge \Xi' \neq \varnothing\} \cup \\ & \{q \mid (\epsilon, q) \in \Xi\} \end{array}$$
   (c) The *induced formula* χ for λ is the conjunction of the formulae obtained in the previous step.
3. The set $\pi_m(\phi)$ is the set of induced formulae for $\lambda \in \Lambda(\mathcal{L})$.

For example, the choice set $\lambda = \{(a, X \cdot b, \neg r), (a, X \cdot b, X)\}$ induces (by step 2) the structural formula $a \Rightarrow \nu X.\,[b]\,(\neg r \wedge X)$. Notice that all induced formulae are closed and guarded whenever the original behavioural one is.

Example 3. Consider formula $\phi = \nu X.\,[a\ \mathsf{call}\ b]\,X \wedge [b\ \mathsf{ret}\ a]\,(\neg r \wedge X)$, *i.e.,* for every program execution consisting of consecutive sequences of calls from a to b followed by a return, the points at which control resumes in a are never return points themselves. Figure 3 shows the mapping $\mathcal{S}$ from the subformulae of ϕ to propositional constants, and the tableau that is constructed for this formula. The first node where a triple is produced is the one labelled ret_1; the triples then propagated to the two leaves that result from application of the rule for atomic propositions, and simple repeat, respectively. The tableau has two leaves with non-empty triple sets; $\mathcal{L}$ thus consists of two sets of two triples each.

Thus, to construct the set of structural formulae, we compute structural formulae for the four choice sets resulting from $\mathcal{L}$:

$$\begin{array}{l} \{(a, X_4 \cdot X_1 \cdot X_2 \cdot b \cdot X_5, \neg r), (a, X_4 \cdot X_1 \cdot X_2 \cdot b \cdot X_5, X_4)\} \\ \{(a, X_4 \cdot X_1 \cdot X_2 \cdot b \cdot X_5, \neg r), (b, X_4 \cdot X_1, \neg r)\} \\ \{(b, X_4 \cdot X_1, \neg r), (a, X_4 \cdot X_1 \cdot X_2 \cdot b \cdot X_5, X_4)\} \\ \{(b, X_4 \cdot X_1, \neg r)\} \end{array}$$

The first set gives rise to the structural formula $a \Rightarrow \nu X_4.\nu X_1.\nu X_2.\,[b]\,\nu X_5.(\neg r \wedge X_4)$, which simplifies to $\chi_1 = a \Rightarrow \nu X.[b](\neg r \wedge X)$: in the text of a, no initial sequence of consecutive call-to-b instructions ends in a return instruction. The

X_0	$\nu X.\,[a\ \mathsf{call}\ b]\,X \wedge [b\ \mathsf{ret}\ a]\,(\neg r \wedge X)$	X_2	$[a\ \mathsf{call}\ b]\,X$	X_4	X	X_6	$\neg r$
X_1	$[a\ \mathsf{call}\ b]\,X \wedge [b\ \mathsf{ret}\ a]\,(\neg r \wedge X)$	X_3	$[b\ \mathsf{ret}\ a]\,(\neg r \wedge X)$	X_5	$\neg r \wedge X$		

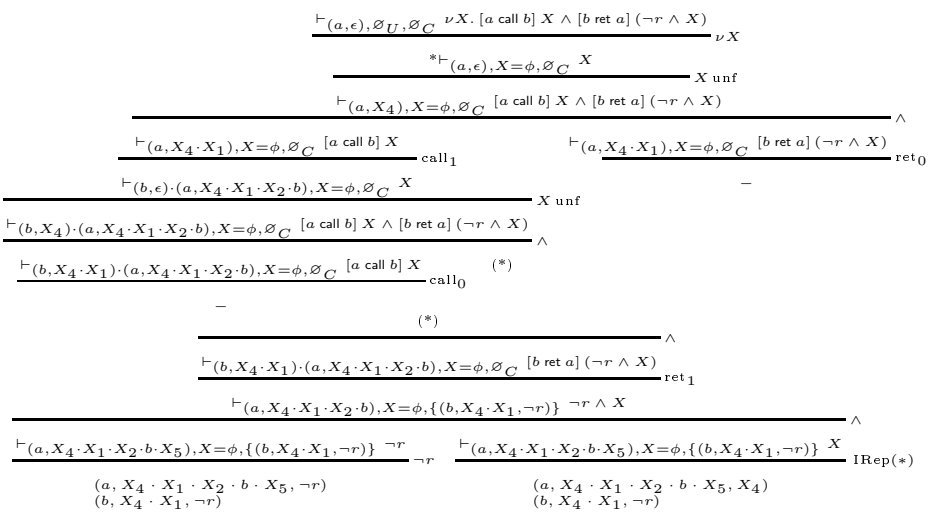

Fig. 3. Tableau for $\nu X.\,[a\ \mathsf{call}\ b]\,X \wedge [b\ \mathsf{ret}\ a]\,(\neg r \wedge X)$ and a, giving rise to $\{a \Rightarrow \nu X.\,[b]\,(\neg r \wedge X), b \Rightarrow \neg r\}$

last set gives rise to the formula $\chi_2 = b \Rightarrow \neg r$ (again after simplification): the text of b does not begin with a return instruction. The formulae constructed from the second and third set are subsumed by χ_2, and hence $\pi_a(\phi) = \{\chi_1, \chi_2\}$. For ϕ and method b there is a single tableau, which has no leaf triples, and hence $\pi_b(\phi) = \{\mathsf{tt}\}$. Thus, $\Pi(\phi) = \{\chi_1, \chi_2\}$.

More property translations and example tableaux illustrating the various repeat conditions can be found in [11].

3.3 Correctness of the Translation

Because of space limitations, we cannot present here the full soundness and completeness proofs; instead, we refer to the accompanying report [11]. The main idea is the construction of a *proof system* that allows to show that a structural formula χ implies a behavioural formula ϕ. The rules of the proof system stand in a one-to-one correspondence with the rules of the tableau system, except for the handling of fixed-point formulae, which are unfolded (so proof trees can be infinite). The proof tree that corresponds to the unfolding of a tableau for behavioural formula ϕ inducing structural formula χ, constitutes a legal proof of χ implying ϕ (*i.e.,* soundness of Π). Moreover, any formula χ' that implies ϕ is subsumed by a formula induced by the tableau (*i.e.,* completeness of Π).

Theorem 2. *Translation Π from behavioural to structural formulae is sound and complete.*

4 Application: Compositional Verification

The original motivation for the present work has been the wish to extend an earlier developed compositional verification method [12] to behavioural properties. The compositional verification method is based on the computation of maximal models: a model is said to be *maximal* for a given property ϕ, if it satisfies ϕ and simulates (*w.r.t.* a property-preserving simulation relation) all other models satisfying ϕ. Due to the close connection between simulation and satisfaction in our logic, we obtain the following compositional verification principle: showing $\mathcal{G}_1 \uplus \mathcal{G}_2 \models \psi$ can be reduced to showing $\mathcal{G}_1 \models \phi$ (*i.e.*, component $\mathcal{G}_1$ satisfies a *local assumption* ϕ) as long as $\mathcal{G}_\phi \uplus \mathcal{G}_2 \models \psi$ (*i.e.*, component $\mathcal{G}_2$, when composed with the maximal flow graph $\mathcal{G}_\phi$ for ϕ, satisfies the *global guarantee* ψ).

Thus, the compositional verification problem is reduced to finding maximal flow graphs. However, given a property ϕ over a flow graph (behaviour), there is no guarantee that the maximal model of ϕ is a valid flow graph (behaviour). At the structural level this problem can be solved, because we can precisely characterise legal flow graphs *w.r.t.* an interface I by a structural formula θ_I in our logic. Then, if ϕ is an arbitrary structural formula, the maximal model of the formula $\phi \wedge \theta_I$ is a flow graph $\mathcal{G}_{\phi,I}$ which represents all flow graphs with interface I that satisfy ϕ.

However, there is no such way to precisely characterise flow graph behaviour in our logic (*cf.* [12]), and thus one cannot directly apply the above compositional verification principle to behavioural properties. In [12], we proposed a "mixed" rule where global guarantees are behavioural, but local assumptions are structural. With the results of the present paper, however, this rule can be combined with the characterisation (1) to yield the following sound and complete compositional verification principle, where both the global guarantee (required to be disjunction-free) and the local assumption are behavioural.

$$\frac{\mathcal{G}_1 \models_b \phi \qquad \left\{\mathcal{G}_{\chi, I_{\mathcal{G}_1}} \uplus \mathcal{G}_2 \models_b \psi\right\}_{\chi \in \Pi_{I_{\mathcal{G}_1}}(\phi)}}{\mathcal{G}_1 \uplus \mathcal{G}_2 \models_b \psi} \quad \mathcal{G}_1 \text{ closed}$$

Notice that when applying the rule, instead of showing $\mathcal{G}_1 \models_b \phi$ it suffices to show $\mathcal{G}_1 \models_s \chi$ for some $\chi \in \Pi_{I_{\mathcal{G}_1}}(\phi)$. Completeness of the principle guarantees that no *false negatives* are possible: if the second premise fails, then there is indeed a legal flow graph $\mathcal{G}$ with interface $I_{\mathcal{G}_1}$ such that $\mathcal{G} \models_b \phi$ but $\mathcal{G} \uplus \mathcal{G}_2 \not\models_b \psi$.

An alternative way of using characterisation (1) for compositional verification is to apply it to the global guarantee (see [11]).

5 Conclusions and Future Work

This paper presents a precise characterisation of (disjunction-free) behavioural formulae as sets of structural formulae, in a context where programs are abstracted as flow graphs, and properties are expressed in a fragment of the modal μ-calculus with boxes and greatest fixed points only. As one significant application, we state a sound and complete *compositional verification* principle for

behavioural properties based on maximal models. Another possible application is the reduction of infinite-state verification of behavioural control flow properties to finite-state verification of structural properties.

Extensions. Unlike the other connectives of the logic, validity of sequents is not compositional *w.r.t. disjunction* in our tableau system. Disjunction can still be handled, though at the expense of completeness, by adding two symmetric tableau rules that simply "drop" the right respectively the left disjunct. A behavioural formula and a method will thus give rise to a set of tableaux, for which we take the union of their induced sets of structural formulae. Alternatively, to potentially obtain a complete translation (if such exists), we plan to generalise the sequent format, *e.g.,* in the style of Gentzen sequents, and then also tableau construction and formula extraction. We also plan to study whether the characterisation can be extended for the logic with diamonds and least fixed points, and for richer program models (*e.g.,* with exceptions, or multithreading, as in [13]), and whether the compositional verification principle can be generalised to *open* components. For the last extension, two different approaches will be considered: (i) the translation is generalised to formulae over open interfaces, requiring the generalisation of Definition 6 for open flow graphs, and (ii) every open component is "closed" by composing it with a *most general environment* before the characterisation is applied.

Implementation. An implementation of the translation has been developed in Ocaml, and is available via a web-based interface [10]. It returns a tableau per method, plus a set of structural formulae (after applying some basic logical simplifications, *e.g.*, removing unused fixed-points, to make the output more readable). It has been applied on all examples in the paper and the accompanying technical report [11]. In all cases, the output is produced within seconds. Various *optimisations* of the translation are possible. For instance, since logically subsumed formulae are redundant in the characterisation, the construction of choice sets can be optimised as follows: if a triple is picked from a contributing leaf, then the same triple must be selected from all other contributing leaves containing it.

In future work, the *complexity* of the tableau construction will be studied by finding upper bounds for the size of generated tableaux, and for the number and size of generated formulae.

References

1. Alur, R., Arenas, M., Barcelo, P., Etessami, K., Immerman, N., Libkin, L.: First-order and temporal logics for nested words. In: Logic in Computer Science (LICS 2007), Washington, DC, USA, pp. 151–160. IEEE Computer Society Press, Los Alamitos (2007)
2. Alur, R., Benedikt, M., Etessami, K., Godefroid, P., Reps, T., Yannakakis, M.: Analysis of recursive state machines. ACM TOPLAS 27, 786–818 (2005)
3. Alur, R., Etessami, K., Madhusudan, P.: A temporal logic for nested calls and returns. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 467–481. Springer, Heidelberg (2004)

4. Bouajjani, A., Fernandez, J.C., Graf, S., Rodriguez, C., Sifakis, J.: Safety for branching time semantics. In: Leach Albert, J., Monien, B., Rodríguez-Artalejo, M. (eds.) ICALP 1991. LNCS, vol. 510, pp. 76–92. Springer, Heidelberg (1991)
5. Burkart, O., Caucal, D., Moller, F., Steffen, B.: Verification on infinite structures. In: Bergstra, J.A., Ponse, A., Smolka, S.A. (eds.) Handbook of Process Algebra, pp. 545–623. North-Holland, Amsterdam (2000)
6. Dam, M., Gurov, D.: μ-calculus with explicit points and approximations. Journal of Logic and Computation 12(2), 43–57 (2002)
7. Dams, D., Namjoshi, K.S.: The existence of finite abstractions for branching time model checking. In: Nineteenth Annual IEEE Symposium on Logic in Computer Science (LICS 2004), pp. 335–344. IEEE Computer Society Press, Los Alamitos (2004)
8. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model checking pushdown systems. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 232–247. Springer, Heidelberg (2000)
9. Grumberg, O., Long, D.: Model checking and modular verification. ACM TOPLAS 16(3), 843–871 (1994)
10. Gurov, D., Huisman, M.: From behavioural to structural properties: A tool web interface, `http://www.csc.kth.se/~dilian/Projects/CVPP/beh2struct.php`
11. Gurov, D., Huisman, M.: Reducing behavioural to structural properties of programs with procedures. Technical Report TRITA-CSC-TCS 2007:3, KTH Royal Institute of Technology, Stockholm, 35 pages (2007), `http://www.csc.kth.se/~dilian/Papers/techrep-07-3.pdf`
12. Gurov, D., Huisman, M., Sprenger, C.: Compositional verification of sequential programs with procedures. Information and Computation 206(7), 840–868 (2008)
13. Huisman, M., Aktug, I., Gurov, D.: Program models for compositional verification. In: International Conference on Formal Engineering Methods (ICFEM 2008). LNCS, vol. 5256, pp. 147–166. Springer, Heidelberg (2008)
14. Huisman, M., Gurov, D.: Composing modal properties of programs with procedures. In: Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA 2007). Electronic Notes in Theoretical Computer Science (to appear, 2008)
15. Kozen, D.: Results on the propositional μ-calculus. Theoretical Computer Science 27, 333–354 (1983)
16. Reddy, U., Kamin, S.: On the power of abstract interpretation. Computer Languages 19(2), 79–89 (1993)
17. Reps, T., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. Science of Computer Programming 58(1-2), 206–263 (2005)
18. Schneider, F.B.: Enforceable security policies. ACM Trans. Infinite Systems Security 3(1), 30–50 (2000)
19. Schöpp, U., Simpson, A.K.: Verifying temporal properties using explicit approximants: Completeness for context-free processes. In: Nielsen, M., Engberg, U. (eds.) FOSSACS 2002. LNCS, vol. 2303, pp. 372–386. Springer, Heidelberg (2002)