

Winskel

Compositional Checking of Satisfaction

HENRIK REIF ANDERSEN

Department of Computer Science, Aarhus University, Ny Munkegade 116, DIC-8000 Aarhus C, Denmark

GLYNN WINSKEL

Department of Computer Science, Aarhus University, Ny Munkegade 116, DIC-8000 Aarhus C, Denmark

Abstract. We present a compositional method for deciding whether a process satisfies an assertion. Assertions are formulas in a modal ν -calculus, and processes are drawn from a very general process algebra inspired by CCS and CSP. Well-known operators from CCS, CSP, and other process algebras appear as derived operators. The method is *compositional in the structure of processes* and works purely on the syntax of processes. It consists of applying a sequence of *reductions*, each of which only takes into account the top-level operator of the process. A reduction transforms a satisfaction problem for a composite process into equivalent satisfaction problems for the immediate subcomponents. Using process variables, systems with undefined subcomponents can be defined, and given an overall requirement to the system, *necessary and sufficient conditions* on these subcomponents can be found. Hence the process variables make it possible to specify and reason about what are often referred to as *contexts*, *environments*, and *partial implementations*. Since reductions are algorithms that work on syntax, they can be considered as forming a bridge between traditional noncompositional model checking and compositional proof systems.

Keywords: process calculi, modal μ -calculus, model checking, compositionality

1. Introduction

In this article we present a compositional method for deciding whether a finite-state process satisfies a specification. Processes will be described in a very general and rich process algebra, which includes common operators from process algebras such as CCS and CSP. This algebra contains primitive operators to reflect sequentiality (by the well-known operation of prefixing), nondeterministic choice, asynchronous and synchronous parallel composition, recursion, relabeling, and restriction. Specifications will be drawn from a modal ν -calculus with negation, in which a variety of properties can be specified. These include the usual *liveness*, *safety*, and *fairness* properties, as well as all operators from ordinary linear and branching-time temporal logics (see, e.g., [1] and [2]).

The method we advocate is *compositional in the structure of processes* and works purely on the syntactical level without any explicit references to the underlying transition system. Compositionality is important for at least the following two reasons. Firstly, it makes the verification *modular*, so that when changing a part of a system only the part of the verification concerning that particular component must be redone. Secondly, when designing a system or *synthesizing* a process, the compositionality makes it possible to have undefined parts of a process and

still be able to reason about it. For instance, it might be possible to reveal inconsistencies in the specification or prove that with the choices already taken in the design, no component supplied for the missing parts will ever be able to make the overall system satisfy the original specification.

This approach is unlike traditional model checking, where a transition system model of a process is built and the specification formula is checked by applying some algorithm to the transition system. There are several versions of this basic idea in the literature, e.g., Emerson and Lei [3], Clarke et al. [4], Stirling and Walker [5], Larsen [6], Winskel [7], Cleaveland [8], and Arnold and Crubille [9]. Recently there have been attempts to extend some of these methods based on transition systems to compositional methods by Clarke, Long, and McMillan [10] and Larsen and Xinxin [11], but none of these are compositional in the structure of processes.

Our method consists of applying a sequence of *reductions*, each of which removes the top-most operator of the process, i.e., a reduction transforms a satisfaction problem for a composite process to satisfaction problems for the immediate subcomponents of the process—without inspecting these. Starting with a process term, one can repeatedly use the reductions until a trivial process (for which satisfaction is easily decided) or a variable remains.

2. The languages

2.1. Syntax

Assume given a set of *state names* Nam , and a finite set of *actions* Act . Processes are denoted by syntactic terms t constructed from the following grammar:

$$t ::= nil \mid at \mid t_0 + t_1 \mid t_0 \times t_1 \mid t \mid \Lambda \mid t\{\Xi\} \mid rec P.t \mid P,$$

where P is an element in Nam , i.e., a state identifier. The usual notion of free and bound will apply to state identifiers P , so that P will be bound in $rec P.t$ but free in $P + nil$.

Nil is the inactive process, and at is the usual prefix and $t_0 + t_1$ the usual sum operations known from CCS. The product term $t_0 \times t_1$ denotes a very general kind of parallel composition that allows the components t_0 and t_1 to proceed both synchronously and asynchronously. The exact semantics is defined below.

A state identifier P in the body of $rec P.t$ works as a *recursion point*, and in effect will behave as the normal recursion in CCS: a term $rec P.t$ has the same behavior as the *unfolded* term $t[rec P.t/P]$ (the result of substituting $rec P.t$ for all free occurrences of P in t). We impose the syntactic restriction on recursive terms, that no product must appear in the body, which ensures that all definable processes are finite state, and for technical reasons we also require every occurrence of P in $rec P.t$ to be *strongly guarded*, i.e., appear immediately under a prefix.

In the prefix at , a denotes an action in Act . For a given set of actions Act , we define a set of *composite actions*. Let $*$ be a distinguished symbol not contained in Act . The symbol $*$ is called the *idling action* and is interpreted as “no action” or “inaction.” Define Act_* to be the least set including $Act \cup \{*\}$ and such that $\alpha, \beta \in Act_*$ implies $\alpha \times \beta \in Act_*$, taking $* \times * = *$. Now $\Xi : Act_* \rightarrow Act$, is a *relabeling* that is a partial function, with finite domain, mapping nonidling actions to nonidling actions. This relabeling can be extended to a total function on Act_* , by taking it to behave as the identity outside the domain. The term $t \upharpoonright A$ is a *restriction* where A is a finite subset of Act_* .

Properties of processes are denoted by assertions A from a modal ν -calculus:

$$A ::= \neg A \mid A_0 \vee A_1 \mid \langle \alpha \rangle A \mid X \mid \nu X.A \mid (t : A),$$

where X ranges over a set of assertion variables. In the maximal fixed-point formula $\nu X.A$, any free occurrence of X must be within an even number of negations in order to guarantee the existence of a unique maximal fixed point. The action name α belongs to the set of composite actions Act_* . The *correctness assertion* $(t : A)$ denotes true if t satisfies A and false otherwise. An assertion is said to be *pure* if it does not contain any correctness assertions.

Many derived operators can easily be defined in terms of the core language and will be used throughout this article:

$$\begin{aligned} [\alpha]A &= \neg \langle \alpha \rangle \neg A, & \mu X.A &= \neg \nu X. \neg A[\neg X/X], \\ T &= \nu X.X, & A \rightarrow B &= \neg A \vee B, \\ F &= \neg T, & A \leftrightarrow B &= (A \rightarrow B) \wedge (B \rightarrow A). \end{aligned}$$

Here we have used the notation $A[B/X]$, which denotes the assertion resulting from substituting B for all free occurrences of X in A . We will say that an assertion A is *closed* if it contains no free variables. Furthermore, for a finite set $K \subseteq Act_*$, we define $\langle K \rangle A = \bigvee_{\kappa \in K} \langle \kappa \rangle A$ where disjunction over an empty set gives false (F).

The correctness assertions $(t : A)$ are atoms in a propositional logic that will be used to express reductions. A grammar for the logic is

$$L ::= T \mid \neg L \mid L_0 \vee L_1 \mid (t : A).$$

In the logical language L , we are able to express complex relationships between properties of different processes. For example,

$$(p + q : \langle \alpha \rangle A) \leftrightarrow (p : \langle \alpha \rangle A) \vee (q : \langle \alpha \rangle A),$$

expresses a very simple example of a reduction. It states that the process $p + q$ can do an α and get into a state that satisfies A if and only if p or q can do an α and get into a state that satisfies A . It is a reduction because the formula is valid for all p 's and q 's, and the validity of $(p + q : \langle \alpha \rangle A)$ is reduced to validity

