Model Checking and Higher-Order Recursion

Hardi Hungar

OFFIS, Oldenburg, Germany
hungar@offis.uni-oldenburg.de

Abstract. Since Muller and Schupp have shown that monadic second-order logic is decidable for context-free graphs in [MS85], several specialized procedures have been developed for related problems, mostly for sublogics like the modal μ-calculus, or even its alternation-free fragment. This work shows the decidability of S1S, the trace version of MSOL, for the richer set of *macro graphs*. The generation mechanism of macro graphs is of higher-order nature and relates to the context-free one like macro grammars [Fis68] relate to context-free grammars.
Technically, the result follows from the decidability of the emptiness problem of the trace language of a macro graph with fairness. The decision procedure is given in form of a tableau system. Soundness and completeness follow from the relation of the (finite) tableaux to their infinite unfoldings. This kind of proof promises to be helpful in the derivation of further results.

1 Introduction

During the eighties several modal logics like CTL, LTL, CTL* and the modal mu-calculus have been developed and shown to be decidable in finite structures. *Model checkers* have been implemented and successfully applied to the verification or debugging of mostly finite systems, e.g. hardware or finite instances of protocols. Infinite-state system, arising from Petri nets or recursive programs, pose a different problem. There have been results for both kinds of systems. Muller and Schupp set a landmark by showing that the monadic second-order logic MSOL of context-free graphs is decidable, though with an inherently nonelementary complexity. What about specialized procedures for weaker logics, for instance the mu-calculus?

Indeed, the nineties have seen several such results so far. Context-free processes[1] are a subset of rooted context-free graphs. Burkart and Steffen were the first to give a decision procedure for the alternation-free mu-calculus in a relevant subset of the context-free processes [BS92], and extended it to the full set of guarded context-free processes in [BS94]. Both generalizations of the structure set and the logic were considered, and local as well as iterative (global) model checkers were developed. One may use a higher-order version generation process for structures and decide the alternation-free mu-calculus [Hun94], or add

[1] The use of the term "context-free" is inconsistent in the literature. We will elaborate later on that. In this short overview, we use our terminology.

M. Kutyłowski, L. Pacholski, T. Wierzbicki (Eds.): MFCS'99, LNCS 1672, pp. 149–159, 1999.

non-synchronized parallel composition like in BPP and check even nonregular properties [BEH95]. Or one may take directly the set of context-free graphs with alternation-free formulas [BQ96], or guarded context-free processes with the full mu-calculus [Wal96,BS97].

The result in this paper is a step towards unification of these results. We show that ω-regular properties, i.e. monadic second-order logic over one successor (S1S) (the trace version of MSOL), can be decided for macro processes. Macro processes were already studied in [Hun94]. They take the idea of a context-free generation to the extreme by permitting (process) variables of arbitrary higher-order type. Ordinary variables appear as objects of type level one. As already known from the study of the call trees of programs with finitely typed procedures, increasing the type level enriches the set of definable structures. Additionally to considering a higher-typed recursion, the assumption of guardedness is dropped, thereby including infinitely branching processes into the picture.

The decidability of ω-regular properties is derived as a consequence of the soundness and completeness of a tableau system deciding the emptiness problem of the trace language of a macro process under an additional fairness constraint. In our construction, the fairness constraint comes from the acceptance set of a Büchi automaton composed in parallel with the macro process at hand.

The tableau system in this paper is a simplified version of one which not only deals with the emptiness problem, but which is also capable of treating arbitrary CTL specifications under fairness constraints.[2] The soundness and completeness proofs for the emptiness problem can be extendeded to the more general case of CTL. The techniques promise to be applicable to further problems in the area. In particular, one can hope to solve the decidability of MSOL in much the same way, extending the result of Muller and Schupp to macro processes. Also, a specialized procedure for the full mu-calculus should be obtainable.

2 Processes and Specifications

2.1 Kripke Structures

Kripke structures and labeled transition systems are equivalent ways of modeling reactive behavior. In this paper, we choose Kripke structures as our semantic domain, and we add fairness constraints.

A *Kripke structure* is a quintuple $(S, R, \mathcal{A}, L, I)$ where S is a set of *states*, $R \subseteq S \times S$ is the *transition relation*, $\mathcal{A}$ is a set of *atoms*, $L : S \to \mathcal{P}(\mathcal{A})$ is the *labeling function*, and $I \subseteq S$ is the set of *initial states*.

Fairness constraints are added as an additional component $C = \{c_1, \ldots, c_n\}$, $c_i \subseteq S$. A *path* in a Kripke structure is an infinite sequence $s_0, s_1, \ldots$ of states with $(s_j, s_{j+1}) \in R$. It is a *fair path* if for all c_i, infinitely many s_j are in c_i. The reader may note that a Büchi automaton is a Kripke structure with finitely many

[2] We had to remove the CTL system from this paper due to space limitations. It was included in the submitted version which can be obtained from the author.

states and one fairness constraint. In this paper, we will consider only structures with one fairness constraint. A state is called fair if it is in the fairness set. A *trace* of a Kripke structure is the sequence of state labelings along a fair path of the structure.

2.2 Macro Processes

We use recursive declarations over *macro terms* for our syntactic representation of infinite Kripke structures. The name "macro" comes from [Fis68], where it was used for a higher-order generalization of context-free grammars. A macro term has a type, which is either κ, the basic type denoting Kripke structures, or it is a functional type $\rho \to \tau$. Each type has a *level* with $\ell(\kappa) = 0$ and $\ell(\rho \to \tau) = \max(1 + \ell(\rho), \ell(\tau))$. The operator "$\to$" is assumed to associate to the right. κ is the only type of level 0, the types of level 1 have the form $\kappa \to \kappa \to \ldots \to \kappa$ with at least two κ .

The set of macro terms over a set of atoms $\mathcal{A}$ is given by

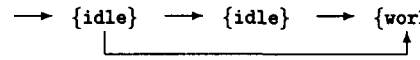
$$\begin{aligned} t_\kappa &::= D \mid t_\kappa + t_\kappa \\ t_{\kappa\to\kappa} &::= \mathtt{A} \\ t_\tau &::= P_\tau \mid t_{\rho\to\tau} \cdot t_\rho \end{aligned}$$

where $\mathtt{A}$ is a finite subset of $\mathcal{A}$ and P_τ is a variable. When writing terms, $\cdot$ is assumed to associate to the left and to have higher priority than $+$. The formal semantics of macro terms without variables is a special case (empty declaration part) of the semantics of a *macro process* provided in Fig. 1. Intuitively, a macro term denotes a finite Kripke structure with an acyclic transition relation. Its states are the subterms of the form $\mathtt{A} \cdot t_\kappa$, which are called *state terms*. Such a term introduces a state labeled by $\mathtt{A}$ which is connected to the initial states of the Kripke structure denoted by t_κ. D (deadlock) stands for the empty structure, $+$ is union. Functional application $(\cdot)$ models, on level one, ordinary sequential composition. Thus, $\mathtt{A}$ can be seen as a state with one "exit", whereas a term of type $\kappa \to \kappa \to \kappa$ gives a Kripke structure with two exits, which are to be connected to the initial state sets of the argument structures.

As an example, the term

$$\{\mathtt{idle}\} \cdot (\{\mathtt{work}\} \cdot D + \{\mathtt{idle}\} \cdot (\{\mathtt{work}\} \cdot D)))$$

denotes a Kripke structure as drawn below.



To get structures with cycles and also infinite structures, we will need recursion. A *recursive declaration* for a set of typed variables $P_1, \ldots, P_n$ is a set of equations

$$P_i \cdot F_{i,1} \cdot \ldots \cdot F_{i,n_i} = t_i(P_1, \ldots, P_n, F_{i,1}, \ldots, F_{i,n_i}) \ ,$$

one for each P_i, where both sides of the equation are of type κ with appropriate typed variables $F_{i,1}, \ldots, F_{i,n_i}$.

A *macro process* is given by a recursive declaration for a set of typed variables $P_1, \ldots, P_n$ and a *main term* $t_\kappa(P_1, \ldots, P_n)$. The maximal type level of the variables gives the *type level* of the process. The Kripke structure denoted by

$$
\begin{array}{ll}
dist_n(D) & =_{df} \emptyset \\
dist_n(\mathtt{A} \cdot t_\kappa) & =_{df} \mathtt{A} \cdot t_\kappa \\
dist_n(t_\kappa + u_\kappa) & =_{df} dist_n(t_\kappa) \cup dist_n(u_\kappa) \\
dist_0(P_i \cdot u_1 \cdot \ldots \cdot u_{n_i}) & =_{df} \emptyset \\
dist_{n+1}(P_i \cdot u_1 \cdot \ldots \cdot u_{n_i}) & =_{df} dist_n(t_i[u_1/F_{i,1}, \ldots, u_{n_i}/F_{i,n_i}]) \\
dist(t_\kappa) & =_{df} \bigcup_{n=0}^{\infty} dist_n(t_\kappa) \\
next(\mathtt{A} \cdot t_\kappa) & =_{df} t_\kappa \\
L(\mathtt{A} \cdot t_\kappa) & =_{df} \mathtt{A} \\
states_0(t_\kappa) & =_{df} dist(t_\kappa) \\
states_{n+1}(t_\kappa) & =_{df} dist(next(states_n(t_\kappa)))
\end{array}
$$

The Kripke structure of a macro process with main term t_0 and occurring set of atoms $\mathcal{A}$ is given by:

$$\left(\bigcup_{n=0}^{\infty} states_n(t_\kappa), dist \circ next, L, \mathcal{A}, dist(t_0)\right)$$

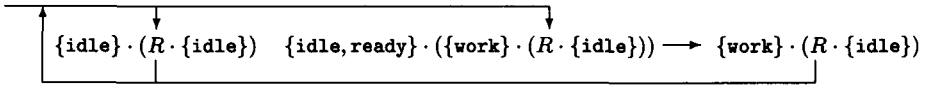
Fig. 1. The semantics of a macro process

a macro process is defined via two auxiliary functions *dist* and *next*. *dist* gives the set of state terms denoted by a term by extracting the summands which are already state terms and inserting the declaration of a variable P for summands of the form $P \cdot t_1 \cdot \ \ldots \ \cdot t_n$, substituting the actual parameters for the formals. *next* provides, for a state term, the term describing its set of successors. The states of the Kripke structure denoted by a macro process are those state terms which can be generated from the main term by repeated application of *dist* and *next*. The composition of *next* with *dist* gives the successor relation. The formal definition is given in Fig. 1. Fairness will be added to the structures by specifying sets of state labelings which are required to occur infinitely often.

As an example of a macro process, consider the declaration

$R_{(\kappa\to\kappa)\to\kappa} \cdot G_{\kappa\to\kappa} \ = \ G \cdot (R \cdot G) \ + \ \{\mathtt{idle}, \mathtt{ready}\} \cdot (\{\mathtt{work}\} \cdot (R \cdot G))$,

with main term $R \cdot \{\mathtt{idle}\}$. This macro process of level 2 defines the finite Kripke structure below.



Though this declaration of type level 2 yields a finite structure, this is not true in general. We will discuss the issue of the power of macro processes in the following.

It is easy to see that we can write any finite Kripke structure as a macro process of type level 0. Just introduce one variable P_s for each state s, and write the equations $P_s = L(s) \cdot (P_{s_1} + \ldots + P_{s_n})$, where the s_i are the successors of s. For the main term, we take the sum of the variables for the initial states. On the other hand, if the process is of level 0, i.e. all variables are of type κ, the denoted Kripke structure is always finite. So type level 0 gives us exactly the finite structures. We call those processes *regular*.

Type level 1 provides us with *more* structures than those which were called context-free processes in [BS92], modulo the fact that we get Kripke structures

instead of labeled transition systems. Already the use of variables of type $\kappa \to \kappa$ yields the recursion pattern available in [BS92]. But we do not have the restriction to *guarded* recursion. This results in the possibility of *infinite branching*, as can be seen from the process with declaration $Q_{\kappa\to\kappa} \cdot F = Q \cdot (A \cdot F) + F$ and main term $B \cdot (Q \cdot D)$. This process denotes a Kripke structure with one initial state labeled by B, which has successors $A^n \cdot D$ for each n.

Furthermore, besides infinite branching, permitting variables with more than one argument of type κ offers the possibility to define the *pushdown processes* of [BS94]. This is most easily seen by resorting to the characterization of pushdown processes as being those structures which result from parallel composition of a finite process and a context-free one (in the sense of [BS92]). In fact, a simple proof shows that on each type level, macro processes are effectively closed under parallel composition with finite structures. The idea is to introduce, for each subterm of the declaration, one copy for each state s of the finite structure representing the original term with the finite structure being in the state s. Transitions in the finite system are represented by an appropriate change of copies. The transformation induces, for m states, m copies of each declared process variable, and in each declaration, m copies of each formal parameter variable. As the type level of a declaration does not depend on the number of parameters but only on their maximal level, this does not increase the type level of the process variables.

Observation 1. *Given a macro process of type level l and a regular process, the synchronous parallel composition of the two is bisimilar to a macro process of level l which can be computed effectively.*

In [BQ96], the problem of model checking *regular graphs* was considered. A graph is regular if it can be generated by a (context-free like) hyperedge replacement system. In fact, in the graph-grammar community they are often called context-free graphs, see [Hab92]. It is easily seen that for every regular graph there is a bisimilar macro process of level 1 (modulo the fact that we use vertex labeling instead of edge labeling which makes no essential difference). The same holds for the context-free graphs from [MS85]. So it can be said that with the exception of the combination of BPP and BPA from [BEH95], all considered domains are subsumed by macro processes of level 1.

As indicated above, the attributes context-free, pushdown and regular are used inconsistently in the literature. From our point of view, we would prefer to use the name context-free process for macro processes of type level 1. These include everything previously named context-free, pushdown or regular in this field. A subset potentially worthwhile to study separately are those processes where recursion is guarded, resulting in structures with bounded degree.

It should be noted that the type level of macro processes induces a strict hierarchy on the set of definable structures. This can be derived from results about the call trees of programs with higher-type procedures [Dam79]. For instance, on type level 2 we can formulate a second-order stack (a stack of stacks), cf. [Hun94]. The traces of a second-order stack cannot be generated by a context-free production system (it is not an algebraic language in the sense of [Tho90]),

while this can be done for each macro process of level 1. In general, a stack of level n [KTU87] can be written as a process of type level n. Besides being able to define more processes in our higher-typed language, there is of course also the benefit of being able to use higher types to better structure the definitions. Going beyond the domain of finitely-typed recursion would, however, be problematic. Processes defined by general, untyped recursion have an undecidable emptiness problem.

2.3 Specifications

As we deal with the emptiness problem, the only specification of a Kripke structure is ε, saying that there is no fair path. Intermediate assertions in a tableau may concern higher-order subterms of macro terms. For those, we introduce higher-order emptiness specifications.

$$\varphi_\kappa ::= \varepsilon, \quad \varphi_{\rho\to\tau} ::= \eta_\rho \to \varphi_\tau, \quad \eta_\rho ::= \emptyset \mid \eta_\rho, \eta_\rho \mid \langle \mathbf{f} \rangle \varphi_\rho \mid \varphi_\rho$$

A higher-order specification ($\varphi_{\rho\to\tau}$) contains a list of assumptions about the argument (η_ρ). This list may be empty ($\emptyset$), or may contain one or more assumptions, which may be guarded. The guard, $\langle \mathbf{f} \rangle$, means that this assumption may be used after a fair state has been encountered, while this is not permitted with unguarded assumptions.

2.4 Sequents

Assertions in the tableau system are sequents of the general form

$$H : t_\tau \vdash \eta_\tau \ .$$

H, if it is not empty, is a list of assumptions about the formal parameters which may appear in t_τ. Elements of the list are sequents of the form $F_\tau \vdash \eta_\tau$.

In the tableau system which is presented in the next section, an assumption not guarded by $\langle \mathbf{f} \rangle$ is replaced by $\emptyset$ when a fair state is encountered explicitly (when reasoning moves from $\mathtt{A} \cdot t_\kappa$ to t_κ, if $\mathtt{A}$ labels a fair state) or implicitly (in the first argument rule, where the presence of $\langle \mathbf{f} \rangle$ implies that inside the process body before the argument was invoked a fair state might have been encountered).

We formulate the maintenance mechanism of fairness guards in an auxiliary function *rem*.

$$\begin{array}{ll} rem(\varphi_\tau) =_{\mathrm{df}} \emptyset & rem(\langle \mathbf{f} \rangle (\varphi_\tau)) =_{\mathrm{df}} \langle \mathbf{f} \rangle \varphi_\tau \\ rem(\emptyset) =_{\mathrm{df}} \emptyset & rem(\eta_1, \eta_2) =_{\mathrm{df}} rem(\eta_1), rem(\eta_2) \\ rem(F \vdash \eta) =_{\mathrm{df}} F \vdash rem(\eta) & rem(H_1, H_2) =_{\mathrm{df}} rem(H_1), rem(H_2) \end{array}$$

We say that a list of assumptions *contains* a sequent, $F_\tau \vdash \varphi_\tau \in H$, if there is a sequent $F_\tau \vdash \eta_\tau$ in H and either φ_τ or $\langle \mathbf{f} \rangle \varphi_\tau$ appears in the list η_τ.

3 The Tableau System

The rules of the system are given in Fig. 2. In general, the schematic rules have the form that one sequent is written above the line, and perhaps more than one

Plus

$$\frac{H : t_1 + t_2 \vdash \varepsilon}{H : t_1 \vdash \varepsilon \,,\, H : t_2 \vdash \varepsilon}$$

Declared call with hypotheses or parameters

$$\frac{H : P \cdot t_1 \cdot \ \ldots \ \cdot t_n \vdash \varepsilon}{P \vdash \eta_1 \to \ldots \to \eta_n \to \varepsilon \,,\, H : t_1 \vdash \eta_1 \,,\, \ldots \,,\, H : t_n \vdash \eta_n}$$

Pure declared call

$$\frac{P \vdash \eta_1 \to \ldots \to \eta_n \to \varepsilon}{F_1 \vdash \eta_1, \ldots, F_n \vdash \eta_n : t_P \vdash \varepsilon} \quad P \cdot F_1 \cdot \ \ldots \ \cdot F_n = t_P \text{ is the declaration of } P$$

Formal call

$$\frac{H : F \cdot t_1 \cdot \ \ldots \ \cdot t_n \vdash \varphi_\tau}{H : t_1 \vdash \eta_1 \,,\, \ldots \,,\, H : t_n \vdash \eta_n} \quad F_\tau \vdash \eta_1 \to \ldots \to \eta_n \to \varphi_\tau \in H$$

Pure constant call

$$\frac{H : \mathtt{A} \vdash \eta \to \varepsilon}{F_\kappa \vdash \eta : \mathtt{A} \cdot F_\kappa \vdash \varepsilon}$$

Constant call with parameter

$$\frac{H : \mathtt{A} \cdot t \vdash \varepsilon}{H : t \vdash \varepsilon} \quad \mathtt{A} \text{ is not a fair labeling} \qquad \frac{H : \mathtt{A} \cdot t \vdash \varepsilon}{rem(H) : t \vdash \varepsilon} \quad \mathtt{A} \text{ is a fair labeling}$$

Argument rules

$$\frac{H : t \vdash \langle \mathbf{f} \rangle \varphi_\tau}{rem(H) : t \vdash \varphi_\tau} \qquad \frac{H : t \vdash \eta_1, \eta_2}{H : t \vdash \eta_1 \,,\, H : t \vdash \eta_2}$$

Fig. 2. The rules for tableau construction

below the line. An *instance* of a rule schema is built by performing appropriate substitutions for the metavariables for specifications and terms.

A tableau is a finite tree built from instances of these rules, starting with $t_\kappa \vdash \varepsilon$ where t_κ is the main term of the macro process to be studied. The intuition about the tableau system is the following. To show that a process has no fair path, we follow all of its paths. Part of the rules formalize the stepwise computation rules for state generation. So, when a declared process variable is encountered, we expand it according to its definition (in the rule *pure declared call*). To avoid having to consider infinitely many terms, we do not substitute the actual parameters for the formals. Instead, we "guess" the specifications of the arguments we will need and add them as hypotheses. The hypotheses are applied using the rule *formal call*, and they are verified in separate branches of the tableau starting below the instance of the pure declared call rule.

It is very important (for completeness) that there can be only finitely many relevant argument properties. The reason for this is that there are only finitely many nonequivalent specifications on each type level. As a consequence, we can expect sequents to repeat which will allow us to produce a finite tableau for those processes having an empty set of fair paths. The top sequents of the instances of the rule *pure declared call* will be the ones we recur to. Which tableaux count as proofs is captured in the definition of a *successful tableau.*

A path in a tableau is *unfair* if it contains neither a node of the form $H : \mathtt{A} \cdot t \vdash \varepsilon$ with a fair labeling $\mathtt{A}$ nor a node $H : t_\tau \vdash \langle \mathbf{f} \rangle \varphi_\tau$. A leaf of the form $P_\tau \vdash \varphi_\tau$ is *recursive* if there is a predecessor with the same sequent and the connecting path is an unfair one. A leaf is *successful* if it is recursive, or of the form $H : t_\tau \vdash \emptyset$ or $H : D \vdash \varepsilon$.

Definition 1. *A tableau is* successful *if all its leaves are successful.*

Plus	$\dfrac{t_1 + t_2 \vdash \varepsilon}{t_1 \vdash \varepsilon \,,\, t_2 \vdash \varepsilon}$	
Process Call	$\dfrac{P \cdot t_1 \cdot \ \ldots \ \cdot t_n \vdash \varepsilon}{t_P[t_1/F_1, \ldots, t_n/F_n] \vdash \varepsilon}$	$P \cdot F_1 \cdot \ \ldots \ \cdot F_n = t_P$ is the declaration of P
Constant call	$\dfrac{\mathbf{A} \cdot t \vdash \varepsilon}{t \vdash \varepsilon}$	

Fig. 3. The rules for the construction of proof trees

Making recursive leaves successful relies on the reasoning that we could go on extending the tableau in always the same way without ever encountering a fair state. Thus, the path we follow in the Kripke structure (if any—there might not be any state term at all on the path in the tableau) is not a fair one. Exactly this argument is used in the soundness proof below.

Before we come to that point, a few remarks on using a tableau system are in order. Proving the absence of a fair path in a macro process could be formulated as an iterative algorithm, and indeed one such algorithm can be derived from the completeness proof. We chose to use a tableau system because other decision procedures, for instance the one for fair CTL, can be formulated more easily that way. And in the same way as the emptiness system generalizes to the one for fair CTL, so do the proofs of its soundness and completeness.

4 Soundness and Completeness

The general idea of the soundness proof is to show that any successful tableau represents a potentially infinite *successful proof tree.* Completeness follows from the existence of a successful proof tree and that each such tree can be folded into a successful (finite) tableau.

Different from tableaux, proof trees do not contain higher-order assertions. They compute on the level of Kripke terms, exploring the structure by applying the functions *dist* and *next*. There is in fact just one proof tree for emptiness. This does not hold for other problems where the general proof technique is also applicable. The proof tree for emptiness is built using the rules from Fig. 3, starting with the same sequent as the tableau. To be successful, the only permitted leaf is $D \vdash \varepsilon$, and the tree must not contain a path with infinitely many fair states.

Lemma 1. *The proof tree of a macro process is successful iff its set of fair paths is empty.*

Soundness of the tableau system follows from:

Proposition 1. *Every successful tableau unfolds into a successful proof tree.*

Proof. (Sketch) To unfold a tableau, we follow its paths, building the proof tree by applying the corresponding proof tree rule as we proceed. When we encounter

a process call, we remember the argument tableaux in an auxiliary environment, to be used when a formal parameter is called. At a recursive leaf, we return to the node it recurs to, usually with an updated environment. That way, a proof tree is generated.

Infinite paths can only be generated by returning from recursive leaves to their predecessors. But the success restriction on recursive leaves guarantees that on the expanded path in the proof tree, no fair state will be encountered between an occurrence of the predecessor and the recursive leaf.

Completeness is a bit more complicated.

Proposition 2. *Every successful proof tree can be folded into a successful tableau.*

Proof. First, we built a possibly infinite tableau corresponding to the proof tree. To do that, we have to extract higher-order assertions and argument assumptions when we encounter a process call. If $P \cdot t_1 \cdot \ \ldots \ \cdot t_n \vdash \varepsilon$ is a node in the proof tree, we take, for each t_i, the set of paths in the tree which lead to a sequent $t_i \cdot \ldots \vdash \varepsilon$. Each such node gives a higher-order assertion about t_i (of lower type level), and the state terms on the connecting path determine whether the fairness guard is added or not. We collect these argument specifications into a list η_i. Remember that there are only finitely many specifications on each type level, so this is always a finite list. We thus get higher-order assertions to be introduced by the rule *declared call with hypotheses or parameters.* The part of the tableau dealing with the process body is generated from the proof tree starting at that node, the argument tableaux are extracted from the respective subtrees.

So we get a possibly infinite tableau, and have to convince us that it contains a successful initial subtree, i.e. that we can cut each infinite path at a successful recurrence. Since there are only finitely many fair states in the proof tree, outside of an initial segment, no fair state does occur. Thus, any repeating higher-order assertion about a declared process variable outside of that initial tree will satisfy the criterion of a successful recursion. And repetitions are bound to occur on each path since there are only finitely many different sequents in the (infinite) tableau.

Full proofs of such (un)folding arguments for related tableau systems are given in [Hun98]. Combining the two propositions, we get our main result.

Theorem 1. *The tableau system is sound and complete.*

Thus, we can construct a finite proof if the set of fair paths of a macro process is empty. To show that the set is *not* empty, one can enumerate sufficiently many tableaux to show that no successful one does exist. It suffices to check all tableaux of a certain maximal depth, for the depth can be bounded by observing that all terms occurring in assertions are subterms of the process declaration and the number of (higher-order) specifications on each type level is finite. This bound is n-exponential for type level n.

An alternative proof could use a dual tableau system establishing the existence of fair paths. In that system, only one branch of a plus is examined, and its

fairness guard requires rather than permits encountering a fair state before using a guarded assumption. Recursions then are successful if a fair state is guaranteed to have been encountered on the path to the recursive leaf.

Remembering that a Büchi automaton is a finite Kripke structure with one fairness constraint, and that macro processes are closed under parallel composition with finite structures (Observation 1), we can conclude the decidability of ω-regular properties for macro processes. These are definable by Büchi automata [Tho90]. We build the product of the complemented Büchi automaton and our macro process and apply the tableaux-based decision method.

Corollary 1. S1S *is decidable for the set of traces of a macro processes.*

5 Conclusion

We have shown that S1S is decidable for macro processes. A context-free graph from [MS85] is (modulo the duality between Kripke structures and edge-labeled graphs) bisimilar to a macro process of level one, while higher type levels generate still more structures, and we do not require a bounded degree of vertices. Thus we have generalized the decidability result from [MS85] to a richer set of graphs, but only for the logic S1S. We conjecture, however, that our proof technique will enable us to derive decidability of an adequate version of MSOL over macro processes, which would result in a true generalization. All that would be needed is an automaton characterization of MSOL over unordered trees, supposedly something like Rabin automata, and an appropriate generalization of the fairness mechanism to the richer form of constraints. Also, we would expect that one could develop a similar tableau system for the full mu-calculus (which is a sublogic of MSOL). We have already done an extension to a system dealing with fair CTL. Tableau systems as ours can of course be turned into iterative, either local or global model-checking procedures. Summarizing, the proof techniques we have applied here seem to be able to cover all of the existing results, with the exception of [BEH95].

[MS85] contains the conjecture that for a graph to have a decidable monadic second-order theory, it must result from a context-free graph Γ by extending Γ, at certain computable points, by one of a finite set of context-free graphs Γ_i. This characterizes a generation process which is of second-order, and, bearing in mind the strictness of the hierarchy induced by the type level, the higher-order result from this paper invalidates this conjecture.

On the other hand, we would not expect to be able to characterize the set of graphs with a decidable monadic second-order theory, nor the set of properties decidable for macro processes. We observe that already in [BEH95] a different extension of the set of processes has been handled (unsynchronized parallel composition), and also nonregular properties were verified, though both the logic and the process language are difficult to relate to other setups. Furthermore, the patterns of macro process definitions stay in the domain of primitive recursion, and we would not suppose decidability to be restricted to that. Stating this

positively, we expect to cover the range of regular properties in finitely-typed recursive systems, and we do believe there is still more to achieve.

Besides the study of reactive systems, a field of applications of the method could be found in the domain of data-flow analysis of recursive programs.

References

[BEH95] A. Bouajjani, R. Echahed, and P. Habermehl, *Verifying infinite state processes with sequential and parallel composition*, 22nd ACM Symp. on Principles of Programming Languages, 1995.

[BQ96] O. Burkart and Y.-M. Quemener, *Model-checking of infinite graphs defined by graph grammars*, Infinity 96, ENTCS 6, Elsevier, 1996.

[BS92] Olaf Burkart and Bernhard Steffen, *Model checking for context-free processes*, 3rd Int. Conf. on Concurrency Theory (R. Cleaveland, ed.), LNCS 630, Springer, 1992, pp. 123–137.

[BS94] Olaf Burkart and Bernhard Steffen, *Pushdown processes: Parallel composition and model checking*, 5th Int. Conf. on Concurrency Theory (B. Jonsson and J. Parrow, eds.), LNCS 836, Springer, 1994, pp. 98–113.

[BS97] Olaf Burkart and Bernhard Steffen, *Model checking the full mu-calculus for infinite sequential processes*, 24th Int. Coll. on Automata, Languages and Programming (Degano, Gorrieri, and Marchetti-Spaccamela, eds.), LNCS 1256, Springer, 1997, pp. 419–429.

[Dam79] W. Damm, *An algebraic extension of the Chomsky-hierarchy*, 8th Conf. on Math. Found. of Comp. Sc. (J. Bečvář, ed.), LNCS 74, 1979, pp. 266–276.

[Fis68] M.J. Fischer, *Grammars with macro-like productions*, 9th Conf. on Switching and Automata Theory, IEEE, 1968, pp. 131–142.

[Hab92] A. Habel, *Hyperedge replacement: Grammars and languages*, LNCS 643, Springer, 1992.

[Hun94] H. Hungar, *Model checking of macro processes*, 6th Int. Conf. on Computer Aided Verification (D.L. Dill, ed.), LNCS 818, Springer, 1994, pp. 169–181.

[Hun98] Hardi Hungar, *Beyond finite-state model checking: Verifying large and infinite systems*, Habilitation, CvO University Oldenburg, 1998.

[KTU87] A.J. Kfoury, J. Tiuryn, and P. Urzyczyn, *The hierarchie of finitely typed functional programs*, 2nd IEEE Symp. on Logic in Comp. Sc., 1987, pp. 225–235.

[MS85] D.E. Muller and P.E. Schupp, *The theory of ends, pushdown automata, and second-order logic*, Theor. Comp. Sc. **37** (1985), 51–75.

[Tho90] Wolfgang Thomas, *Automata on infinite objects*, Handbook of Theoretical Computer Science (J. van Leeuwen, ed.), Elsevier, 1990.

[Wal96] I. Walukiewicz, *Pushdown processes: games and model-checking*, 8th Int. Conf. on Computer Aided Verification, LNCS 1102, Springer, 1996.