Model Checking of Macro Processes

Hardi Hungar

Computer Science Dept.,
University Oldenburg,
D-26111 Oldenburg, Germany
hungar@informatik.uni-oldenburg.de

Abstract.
Decidability of modal logics is not limited to finite systems. The alternation-free modal mu-calculus has already been shown to be decidable for context-free processes, with a worst case complexity which is linear in the size of the system description and exponential in the size of the formula. Like context-free processes correspond to the concept of procedures without parameters, *macro* processes correspond to procedures with procedure parameters. They too allow deciding mu-calculus formulae, as is shown in this paper by presenting both global (iterative) and local (tableaux-based) procedures. These decision procedures handle correctly also process systems which are defined by unguarded recursion. As expected, the worst case complexity depends on the highest type level in the process description, and it is k-exponential in the size of the formula for a system description with type level k.

1 Introduction

Model checking provides a powerful tool for the automatic verification of behavioral systems. The corresponding standard algorithms fall into two classes: the iterative algorithms (cf. [15, 8, 11, 12]) and the tableaux-based algorithms (cf., e.g. [3, 4, 9, 30, 33, 35]). Whereas the former class usually yields higher efficiency in the worst case, the latter allow *local* model checking (cf. [33]), which avoids the investigation of for the verification irrelevant parts of the process being verified.

At first sight both kinds of algorithms seem to be restricted to finite systems, although Bradfield and Stirling [3, 4] constructed a sound and complete tableau system for the full mu-calculus [28], which can deal with *infinite* transition systems: In general, neither the assertions in the tableaux are purely syntactic nor is well-formedness of a tableau decidable. But by using second-order assertions Burkart and Steffen [5] were able to handle infinite systems given in the form of *context-free* process systems with an iterative algorithm, and a local model checking procedure has been developed by Hungar and Steffen [24]. Also, an automata-based approach has been presented by Iyer [26]. This one does not decide mu-calculus formulae but checks whether the process has a computation which is accepted by a given Büchi automaton.

Context-free processes (CFPs) are essentially equivalent to BPA processes [2]. They are labeled transition systems generated by edge replacement systems: edges labeled with a nonatomic action (a procedure) are to be replaced by the transition system defining the action (the body of the procedure). Because the defining systems may

contain nonatomic actions as well, the resulting system might be infinite. But for model checking purposes, only the effect of an action to the validity of subformulae of the formula in question needs to be observed. This allows the reduction of the problem to a finite mutual recursive equation system, which was the central idea of [5, 24].

CFPs can model some interesting infinite structures, for example a stack. They can not, however, model a queue. This should not come as a surprise, because a queue would give the power of a Turing-machine, destroying decidability. Also, two stacks in parallel would yield undecidability. Therefore, CFPs are not closed under parallel composition [6]. But there are other ways in which the expressibility can be increased without destroying decidability of mu-calculus formulae. And this is done within this paper.

Just like context-free grammars can be generalized to macro grammars [17], and further to higher-typed production systems, capable of generating more and more languages [13, 16], one can also study higher-typed expansion rules for the generation of processes, resulting in what I call *macro processes.* In programming language terms, macro processes (MPs) correspond to programs with higher-order procedures, whereas context-free processes correspond to programs with parameterless procedures. Another way of introducing macro processes would be to enrich BPA by process abstraction and application.

Recalling the results of classical program verification [18, 19], one could expect to be able to lift the algorithms working for CFPs to the macro case. And indeed this is the main result of this paper. Already the automata-based approach from [26] can cope with macro processes of second order (only parameterless processes are allowed as parameters). Here, arbitrary finite types are allowed. The price when dealing with MPs is that the upper bound for the worst-case complexity of the algorithms is k-exponential in the size of the formula, where k is the maximal type level in the process system definition. This hyperexponential growth is again in accordance with similar results for higher-order programming languages [22, 23, 29]. It is a worst-case complexity, though, and an efficient implementation should usually behave much better.

The new algorithms also cope with unguarded recursion, which allows to define processes with infinite branching. In that respect they generalize the known algorithms even for unparameterized processes. For parameterized processes, this is of particular interest, because it is much harder to check finite branching for parameterized processes than for unparameterized (context-free) processes.

The next section introduces the technical definitions and demonstrates the modeling power of MPs by giving an example. Then the model checking algorithms are presented, and in the final section extensions and open questions are discussed.

2 Processes and Formulae

In this section I introduce *process graphs* as the basic structure for modeling behavior, and more specifically, *macro process systems* as finite representations of infinite process graphs, as well as the (alternation-free) modal mu-calculus as a logic for specification.

2.1 Macro Process Systems

Definition 1 (Process Graphs). A *process graph* (short *PG*) is a labeled transition system G with distinguished start and end state (s_G and e_G, $s_G \neq e_G$), where no transition originates at e_G. I.e. it consists of a set (of states), a binary relation on the states (transitions), and a labeling of the transitions.

Intuitively, a process graph encodes the operational behavior of a process. That the end state of a process graph must not be the origin of a transition has its reason in the use which will be made later of process graphs: They will replace edges in transition systems, and that an inserted process graph may be entered via its end state has to be avoided. Leaving the inserted process graph via the start state is avoided by using a specific form of insertion, called *embedding*.

Definition 2 (Embedding). Let G and G' be PGs and let s_1 and s_2 be states in G'. *Embedding G into G' between s_1 and s_2* produces a PG where a copy of G is inserted into G' with

- all states in the copy are new except the end state, for which s_2 is used,
- for all edges leaving s_G, an edge with the same label and destination originates at s_1.

The copy of s_G is called the *initial state* of the embedding and s_2 is called its *return state*.

I want to remark that the construction ensuring that an embedded PG is not left through the start state could also have been used for the end states, to eliminate the restriction on transitions leaving end states in Definition 1. This choice is a matter of taste and of technical significance only.

To generalize the notion of a context-free process system, I first introduce *finite types*.

Definition 3 (Finite Types). The set of *finite types* is given by the production rules

$$\tau ::= \beta \mid (\tau \rightarrow \tau).$$

β is the base type.

The *level* of a type is inductively defined by:

$$\mathrm{lev}(\beta) = 0, \quad \mathrm{lev}(\tau \rightarrow \rho) = max(\mathrm{lev}(\tau) + 1, \mathrm{lev}(\rho)).$$

For every type τ, an infinite set of identifiers X_τ is available, with $x_\tau, p_\tau \in X_\tau$. *Typed terms* are built up from identifiers by type-respecting application, i.e.

$$t_\tau ::= x_\tau \mid (t_{\tau \rightarrow \tau}\, t_\tau)$$

Simple finite types are generated from $\beta \rightarrow \beta$ instead of β. Accordingly, *simple identifiers* are those of simple types, and *simple terms* do only have subterms of simple type.

Apart from the notions of *simple* types and terms, the definition is fairly standard. Note that types of level 1 are just curried forms of the usual first-order types $\beta \times \cdots \times \beta \rightarrow \beta$. I use an ML-style way of writing applications, i.e. $(p\ x\ y)$ means that the result of p applied to x is applied to y.

Simple terms of type $\beta \rightarrow \beta$ will be used as nonatomic actions in the following definition of macro processes. Unrestricted finite types are considered in the discussion of extension in the final section.

Definition 4 (Macro Processes). Let *Act* be a set of *atomic actions.*

A *macro process system* (MPS) has the form

$$\begin{aligned} (p_1\ x_{1\,1}\ \ldots\ x_{1\,n_1}) &= G_{p_1} \\ &\ldots \\ (p_k\ x_{k\,1}\ \ldots\ x_{k\,n_k}) &= G_{p_k} \\ &\quad G_{main}\ . \end{aligned}$$

It consists of a set of *process definitions* defining process identifiers $p_1, \ldots, p_k$ and a *main process graph* G_{main}. The process graphs $G_{p_1}, \ldots, G_{p_k}$ and G_{main} have start states $s_{p_1}, \ldots, s_{p_k}$ and s_{main} and end states $e_{p_1}, \ldots, e_{p_k}$ and e_{main}. Labels in the graphs are either atomic actions or simple terms of type $\beta \rightarrow \beta$. In addition to the defined identifiers, which may occur in any of the process graphs, the terms in G_{p_i} may contain $x_{i\,1}, \ldots, x_{i\,n_i}$, i.e. the formal parameters of the definition. The *level* of an MPS is the highest occuring type level.

The level of an MPS is the maximum of the levels of its defined identifiers. Context-free process systems coincide with the set of MPSs of level one.

I will give a simple *operational* semantics for an MPS, by defining its *complete expansion.* This results from the main PG by repetetively replacing all nonatomic actions by their definitions. The complete expansion may be infinite or even infinitely branching.

Definition 5 (Complete Expansion). *Expanding* a nonatomic transition $s_1 \stackrel{(p_i\ t_1\ \ldots\ t_n)}{\longrightarrow} s_2$ in the main PG of an MPS embeds G_{p_i}, the defining PG of p_i, with the formal identifiers $x_{i\,j}$ in transition labels replaced by the t_j, between s_1 and s_2 and removes the transition $s_1 \stackrel{(p_i\ t_1\ \ldots\ t_n)}{\longrightarrow} s_2$.

An *expansion step* replaces all nonatomic edges in parallel. The *complete expansion* of an MPS results by repeated application of expansion steps. I.e. the complete expansion is the union of a (countable) chain of approximants, where an approximant is the PG of the MPS after a finite number of expansion steps with all nonatomic edges deleted.

The *finite language* of an MPS consists of those strings of atomic actions which label finite paths from the start to the end state of the complete expansion.

Note that expanding an edge in an MPS always yields a well-formed MPS. The result of an expansion step is unique up to renaming of states, thus also the complete expansion is uniquely determined.

2.2 An Example of an MPS

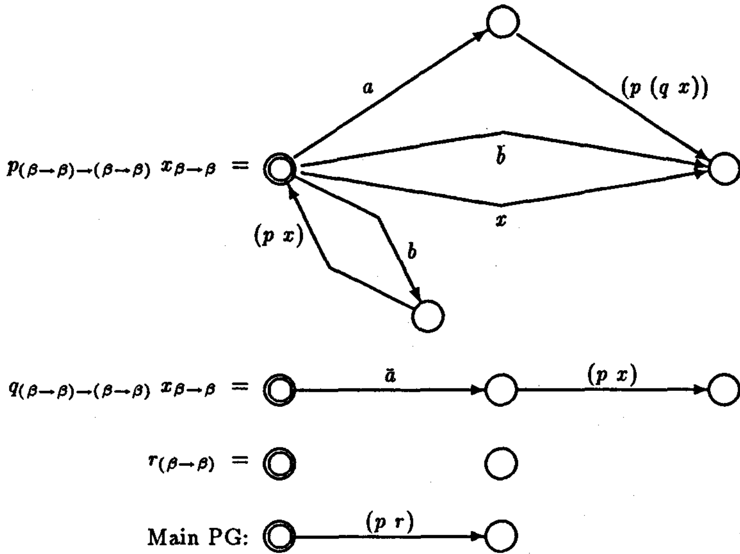

Fig. 1. MPS for a 2-stack.

Figure 1 shows the MPS for a 2-stack. There are several different, but equivalent versions of n-stacks [27]. Here, a 2-stack is a stack of ordinary stacks over a set of basic elements with (partial) operations

- $\mathbf{push_1(d)}$: push $\mathbf{d}$ on the top ordinary stack
- $\mathbf{pop_1}$: pop the top ordinary stack
- $\mathbf{push_2}$: duplicate the top stack
- $\mathbf{pop_2}$: remove the top stack ,

which initially contains an empty ordinary stack.

In the MPS of Figure 1, a and $\bar{a}$ are the actions for $\mathbf{push_1(d)}$ and $\mathbf{pop_1}$, and b and $\bar{b}$ are used for $\mathbf{push_2}$ and $\mathbf{pop_2}$. This MPS abstracts from the differences between the basic elements and just keeps their count, but it is obvious how to change it in order to model a 2-stack over a given finite set precisely (For each basic element, an atomic action for pushing and popping would be introduced). Also nested stacks of higher degree can be modeled (by MPSs of higher degrees).

A formal argument that the given MPS indeed models a 2-stack would not be hard to give. One can establish a bisimulation between a 2-stack and the MPS. Roughly, the correspondence is as follows. During execution, the parameter of the actual instance of p represents the top 1-stack. It is always of the form $(q^n\ r)$, which

allows n consecutive a actions and thus corresponds to a 1-stack with n elements. A b starts a new recursive call with the same parameter, after which the current call is continued (i.e. it duplicates the top 1-stack), whereas a $\bar{b}$ returns to the previous incarnation of p (i.e. it removes the top 1-stack).

Nested stacks are outside the scope of context-free processes. One could use the pumping lemma to show that the finite language of a nested stack (of nesting depth greater than one) is not context-free. Indeed, in several respects an n-stack comprises the essence of computation with procedures of type level n, cf. [14, 29], which in their computation power form an infinite hierarchy.

2.3 Mu Calculus

The following negation–free syntax defines a sublanguage of the mu-calculus, which in spite of being as expressive as the full mu-calculus allows a simpler technical development.

$$\phi ::= \mathit{ff} \mid \mathit{tt} \mid X \mid \phi \wedge \phi \mid \phi \vee \phi \mid [a]\phi \mid \langle a\rangle\phi \mid \nu X.\phi \mid \mu Y.\phi$$

In the above, $a \in Act$, and $X, Y \in Var$, where Var is a set of variables. Only *closed*, *alternation free* formulae will be used, i.e. every variable is bound by a ν or μ, and no ν-subformula has a free variable which, in the context of the whole formula, is bound by a μ, and vice versa. The set of closed alternation-free formulae is denoted by $\mathcal{F}$.

Given a PG, the semantics of a formula is a subset of its states. I do not give the formals of the standard definition. The *closure* $CL(\phi)$ of a formula ϕ is the set of all its subformulae with each free variable replaced (iteratively) by the corresponding fixpoint subformula, e.g. $CL(\nu X.\phi) = \{\nu X.\phi\} \cup CL(\phi[\nu X.\phi/X])$.

2.4 Higher-Order Formulae

Definition 6 (Higher-Order Formulae).
Let the set of *typed higher-order formulae* be defined by

$$\mathcal{F}_\beta =_{\mathrm{df}} \mathcal{F}$$
$$\mathcal{F}_{\tau\to\tau} =_{\mathrm{df}} \mathcal{P}(\mathcal{F}_\tau) \times \mathcal{F}_\tau \ ,$$

where $\mathcal{P}$ denotes the powerset of a set. A higher-order formula (HOF) is *finite* iff its components are finite. The *basis* of a higher-order formula is the set of all its constituent mu-calculus formulae, i.e. $(b\ \phi_\beta) = \phi_\beta$, $(b\ (\Theta, \phi)) = \{(b\ \theta) \mid \theta \in \Theta\} \cup \{(b\ \phi)\}$.

The relation $\preceq$, defined by

$$\phi_\beta \preceq \psi_\beta =_{\mathrm{df}} \phi_\beta = \psi_\beta$$
$$\Phi_\tau \preceq \Psi_\tau =_{\mathrm{df}} \forall\phi_\tau \in \Phi_\tau . \exists\psi_\tau \in \Psi\tau . \phi_\tau \preceq \psi_\tau$$
$$(\Phi_\tau, \phi_\tau) \preceq (\Psi_\tau, \psi_\tau) =_{\mathrm{df}} \Psi_\tau \preceq \Phi_\tau \ \wedge\ \phi_\tau \preceq \psi_\tau \ ,$$

gives a syntactic approximation of implication on higher-order formulae.

Higher-order formulae will be denoted by lowercase greek letters, whereas uppercase letters will be used for sets of formulae.

The objects denoted by higher-order formulae are terms over the defined identifiers of an MPS. The base case of the definition concerns the validity of second-order assertions for arbitrary PGs, and this is explained as ordinary validity for embeddings of the PG.

Definition 7 (Semantics of Higher-Order Formulae). Let G be a PG. Then $G \models (\Theta_\beta, \phi_\beta)$ iff the initial state of every embedding of G satisfies ϕ_β, whenever the return state satisfies all formulae in Θ_β.

Let $t_{\beta\to\beta}$ be a term of defined identifiers in an MPS. $t_{\beta\to\beta} \models \phi_{\beta\to\beta}$ iff the complete expansion of $s \overset{t_{\beta\to\beta}}{\to} e$ satisfies $\phi_{\beta\to\beta}$.

For higher simple types $\bar{\tau} \to \tau$, $t_{\bar{\tau}\to\tau} \models (\Theta_{\bar{\tau}}, \phi_\tau)$ iff, for all $t_{\bar{\tau}}$ with $t_{\bar{\tau}} \models \Theta_{\bar{\tau}}$, it holds that $(t_{\bar{\tau}\to\tau}\ t_{\bar{\tau}}) \models \phi_\tau$ (the MPS may be extended to define identifiers in $t_{\bar{\tau}}$).

The usefulness of higher-order formulae for the verification relies on two observations:

- For verification purposes, defined processes are completely determined by the set of formulae they satisfy, which is important for the completeness of compositional reasoning.
- If the validity of a finite formula is concerned, only finite formulae need to be considered, which will make the method effective.

In other words: a finite set of formulae is an adequate abstraction of the true semantics of a macro process. This is formalized in the following proposition.

Proposition 8 (Adequacy).

1. *Let G be a PG, and let s_i resp. s_r be the initial resp. return state of an embedding of G. Then*

$$s_i \models \phi_\beta \quad \text{iff} \quad G \models (\{\theta \in CL(\phi_\beta) \mid s_r \models \theta\}, \phi_\beta) .$$

2. *Let $t_{\bar{\tau}\to\tau}$ and $t_{\bar{\tau}}$ be terms of defined identifiers. Then*

$$(t_{\bar{\tau}\to\tau}\ t_{\bar{\tau}}) \models \phi_\tau \text{ iff } t_{\bar{\tau}\to\tau} \models (\{\theta_{\bar{\tau}} \in \mathcal{F}_{\bar{\tau}} \mid t_{\bar{\tau}} \models \theta_{\bar{\tau}} \text{ and } (b\ \theta_{\bar{\tau}}) \subseteq CL(b\ \phi_{\bar{\tau}\to\tau})\}, \phi_\tau) .$$

Formulae of type $\beta \to \beta$ are essentially the second-order formulae from [24]. Intuitively, $(\Theta_\beta\ , \phi_\beta)$ is true if the start state always satisfies ϕ_β whenever at the end state a PG is added, so that the end state now satisfies each formula in Θ_β. Higher-order formulae impose this on the PG if all parameters satisfy their specification.

3 Model Checking of Macro Processes

I will formulate the algorithms in this section w.r.t. a given MPS with defined processes $p_1, \ldots, p_k$.

3.1 Local Model Checking

The local method is tableaux-based. I give a set of tableau rules. Other than the iterative algorithm to be presented in the following subsection, local model checking allows to ignore irrelevant parts of the MPS.

Intermediate formulae in the proofs concern either states of defining PGs with appropriate assumptions about the parameters and the return state, or terms which might be called or passed as parameters in calls. Such an intermediate formula is called a *sequent*.

Definition 9 (Sequents). A *(higher-order) assertion* is either a *state assertion* s **sat** ϕ_β or a *term assertion* t_τ **sat** ϕ_τ for a simple type τ, with finite ϕ in both cases.

A *hypotheses set* Γ_{p_i} for a defined identifier p_i is a set of assertions of the form

$$\{e_{p_i} \textbf{ sat } \phi_0,\ x_{i\,1} \textbf{ sat } \phi_1,\ \ldots\ , x_{i\,n_i} \textbf{ sat } \phi_{n_i}\}\ ,$$

i.e. a collection of assertions containing one assertion for each parameter and one for the end state of the definition.

A *sequent* is either $\Gamma_{p_i} \vdash s$ **sat** ϕ_β where s is one of the states of p_i or $\Gamma_{p_i} \vdash t_\tau$ **sat** ϕ_τ where t_τ only involves defined identifiers and formal parameters of p_i .

The validity of sequents is defined similar to the validity of formulae: The assertion must hold whenever the formal parameters are instantiated with defined identifiers satisfying the hypotheses.

The rule set is an adaptation of the rule set from [24], by adding higher-type reasoning like in [19].

Definition 10 (Successful Tableau, Derivability). A tableau is *successful* if it is finite and all *leaves* are *successful*. *Successful leaves* are of the form

1. $\Gamma \vdash s$ **sat** *tt*, or
2. $\{\ldots\ x_\tau$ **sat** $\Theta_\tau\ \ldots\} \vdash x_\tau$ **sat** θ_τ where $\{\theta_\tau\} \preceq \Theta_\tau$, or
3. $\Gamma \vdash s$ **sat** $\phi(\nu X.\psi)$, where $\phi(\nu X.\psi) \in CL(\nu X.\psi)$, there is another node on the path from the root of the tableau to this node labeled with the same sequent, and the maximal fixpoint gets unfolded between these two nodes, or
4. $\Gamma \vdash p$ **sat** $(\Theta_{\tau_1}, \ldots, (\Theta_\beta, [a]\phi_\beta)\ldots)$, where the assertion recurs along a path where the last component of the formula is always $[a]\phi_\beta$.

A sequent is *derivable* if it has a successful tableau.

A node in a tableau may also have no successor without being a leaf. This applies to sequents of the form $\Gamma \vdash s$ **sat** $[a]\phi_\beta$, if neither a-transitions nor nonatomic transitions originate at s, for example if $s = e_{main}$.

Some remarks in order to explain the mechanism of tableau construction: At any time, reasoning is restricted to one process definition (resp., the main process), and the hypotheses contain a set of assertions about each parameter and the end state of the definition. A subtableau for a process call is entered only when necessary, i.e. when evaluating a modality at the origin of a nonatomic transition. Then, ultimately

Main

$$\frac{H \text{ sat } \phi_\beta}{\{\} \vdash s_{main} \text{ sat } \phi_\beta}$$

Basic

$$\frac{\Gamma \vdash s \text{ sat } \phi_\beta \wedge \psi_\beta}{\Gamma \vdash s \text{ sat } \phi_\beta \quad \Gamma \vdash s \text{ sat } \psi_\beta} \qquad \frac{\Gamma \vdash s \text{ sat } \phi_\beta \vee \psi_\beta}{\Gamma \vdash s \text{ sat } \phi_\beta} \qquad \frac{\Gamma \vdash s \text{ sat } \phi_\beta \vee \psi_\beta}{\Gamma \vdash s \text{ sat } \psi_\beta}$$

$$\frac{\Gamma \vdash s \text{ sat } [a]\phi_\beta}{\Gamma \vdash s' \text{ sat } \phi_\beta \quad \ldots \quad \Gamma \vdash s'' \text{ sat } \Theta_\beta \quad \Gamma \vdash t_{\beta\to\beta} \text{ sat } (\Theta_\beta\,,\, [a]\phi_\beta)}$$

(All s' where $s \xrightarrow{a} s'$ and all $t_{\beta\to\beta}$, s'' where $s \xrightarrow{t_{\beta\to\beta}} s''$. $s \notin \{\, s_p \mid p \text{ defined identifier} \,\}$).

$$\frac{\Gamma \vdash s \text{ sat } \langle a\rangle\phi_\beta}{\Gamma \vdash s' \text{ sat } \phi_\beta}\ (s \xrightarrow{a} s') \qquad \frac{\Gamma \vdash s \text{ sat } \langle a\rangle\phi_\beta}{\Gamma \vdash s'' \text{ sat } \Theta_\beta \quad \Gamma \vdash t_{\beta\to\beta} \text{ sat } (\Theta_\beta\,,\, \langle a\rangle\phi_\beta)}\ (s \xrightarrow{t_{\beta\to\beta}} s'')$$

$$\frac{\Gamma \vdash s \text{ sat } \nu X.\phi_\beta}{\Gamma \vdash s \text{ sat } \phi_\beta[\nu X.\phi_\beta/X]} \qquad \frac{\Gamma \vdash s \text{ sat } \mu Y.\phi_\beta}{\Gamma \vdash s \text{ sat } \phi_\beta[\mu Y.\phi_\beta/Y]}$$

Higher Types

$$\frac{\Gamma \vdash (t_{\tau\to\tau}\ t_\tau) \text{ sat } \phi_\tau}{\Gamma \vdash t_{\tau\to\tau} \text{ sat } (\Theta_\tau, \phi_\tau) \quad \Gamma \vdash t_\tau \text{ sat } \theta_\tau,\ \theta_\tau \in \Theta_\tau}$$

$$\frac{\Gamma \vdash p \text{ sat } (\Theta_{\tau_1}, (\Theta_{\tau_2}, \ldots (\Theta_\beta, \phi_\beta)\ldots))}{\{x_1 \text{ sat } \Theta_{\tau_1}\,,\, x_2 \text{ sat } \Theta_{\tau_2}\,, \ldots,\, e_p \text{ sat } \Theta_\beta\} \vdash s_p \text{ sat } \phi_\beta}$$

Fig. 2. Tableau Rules

a modality formula about the initial state of another process has to be proven. This depends only on the successors of the initial state, which makes this reasoning sound, considering the construction of the complete expansion.

Since all rules are (backwards) sound, the soundness of the method follows from the validity of each successful leaf. Both forms of recurring leaves require an argument. The recurrence of a maximal formula by unfolding the maximal fixpoint on the connecting path suffices (in the alternation-free mu-calculus) for an application of the maximal fixpoint characterization. Note that the maximal fixpoint formula need not recur itself, because it might only appear in disjoint subtableaux handling process calls (compare the example in [24]). If a p sat $(\Theta_{\tau_1}, \ldots, (\Theta_\beta, [a]\phi_\beta)\ldots)$ recurs without the box being removed on the path, all possibilities of invalidating the assertion by doing an a-step are examined in subtableaux along the path. The minimality of the complete expansion guarantees that this is sufficient.[1]

The completeness relies on the finiteness of the set of relevant assertions and sequents, and on the adequacy of the higher-order semantics. To prove a formula ϕ_β,

[1] Box recurrence is an effect of allowing unguarded recursion in MPSs.

the basis of intermediately needed higher-order formulae is in $CL(\phi_\beta)$ (finiteness). For every valid ν-formula, some sequent with an element of its closure must recur, if the rules are applied with care (adequacy). And a valid μ-formula must have a finite justification.

The finiteness accounts also for the completeness of an exhaustive search. Summing up, we have:

Theorem 11. *The tableau system provides an effective, sound and complete model checking procedure for macro processes and alternation-free formulae.*

3.2 Iterative Model Checking

The iterative algorithm follows the idea of [5], only that it is extended to macro processes and that it copes with infinitely branching processes (generated by unguarded recursion).

Since the formulae are alternation-free, minimal and maximal fixpoints can be treated separately. To check the validity of a formula ϕ for the start state (of the complete expansion), the iterative algorithm essentially computes all valid sequents. To be more precise, if s is a state of p_i and a complete semantic description of the parameters and the end state of p_i is given, the subset of $CL(\phi)$ which is valid at s is computed. The semantic description of an end state is a subset of $CL(\phi)$ (the subformula which are assumed to be valid). For a parameter, a monotonic function over $\mathcal{P}(CL(\phi))$ of appropriate type is taken as description. For identifiers of $\beta \rightarrow \beta$, this is just the second-order semantics of [5]: The function yielding the set of formulae valid at the start state, given the set of formulae at the end[2]. For higher types, it is a higher-order monotonic function. The function (of parameter and end state descriptions) computed for the s_{p_i} gives the semantic description for p_i, the result for s_{main} gives the set of formulae valid for the MPS.

I will sketch the computation step for minimal fixpoint operators. It is best explained for the case of one fixpoint only. Every fact about subformulae not containing the fixpoint variable can be assumed to be known. Let Ψ be the set of closure formulae containing the fixpoint formula. The computation iteratively approximates from below the subsets of Ψ valid at the states of the definitions for all possible parameter descriptions. It starts with the end states of process definitions initialized according to the respective description values, and everything else initialized to the empty set. Then, modalities and conjunctions and disjunctions are evaluated, and fixpoints may be unfolded. Transitions labeled with atomic actions are easy to cope with, transitions labeled with terms need an evaluation of the term according to the parameter descriptions (for parameters of the definition) respectively the current aproximation of the higher-order semantics of a defined identifier. This computation is monotonic, i.e. the sets of valid formulae do increase[3]. Therefore, it will reach a fixpoint, which itself gives a monotonic function for each state.

[2] This description of the algorithm is still oversimplified. E.g. in case that the argument set is inconsistent, the result of the computation of the algorithm need not be the full set $CL(\phi)$. In general, only a subset is computed. But the result is correct if the argument set consists of all subformulae valid for some process graph. A similar remark applies to parameter specifications.

[3] This relies on the monotonicity of the functions for parameter descriptions.

If recursion was always guarded, this fixpoint would be the correct approximation of the semantics of the definitions (as in [5]). In the presence of unguarded recursion, it has to be checked whether each box formula $[a]\theta$ which is computed as *ff* is properly invalidated by some other formula, i.e. whether according to the parameter descriptions there is a transition labeled with a to a state where θ is false. This will always be the case except when a defined process recursively calls itself (with semantically equivalent parameters) without any atomic action occuring before the second call. If some box formulae are set to *tt* because they do not pass the test, the iteration continues until finally a fixpoint is reached where all false box-formulae are properly invalidated.

The computation process for maximal fixpoint formulae is the dual procedure. It approximates the result from above, and has a special check for the validity of diamond formulae.

Once the higher-order descriptions of the defined identifiers are computed, it is easy to decide whether the formula in question holds at the start state of the main process graph.

The complexity of this iterative computation is dominated by length of representations of the descriptions of defined identifiers. This representation is k-exponential in $O(lev(\tau))$ for an identifier of type τ. I.e. for context-free processes the algorithm is exponential, for second-order processes it is double exponential, and so on.

4 Extensions and Open Problems

The presented model checking procedures are restricted to MPSs with simple types, i.e. the type hierarchy in fact starts with $\beta \to \beta$. It is, however, quite straightforward to extend them to arbitrary finite types. But what should be a process of type $\beta \to (\beta \to \beta)$? The answer is simple: The argument of type β of a simple process is instantiated with the return state of the call, so an extra argument of this type should be regarded as a second possible return state. A call to this process would have to be represented by a *hyperedge* with one origin and two destinations, and the defining process graph would have *two* end states.

Hyperedge replacement systems [21] are, anyhow, a more adequate generalization of context-free string grammars to graph grammars than edge replacement systems. They can generate all [7] graphs of pushdown automata [32] (exactly the same set of finitely branching graphs). In [6], the iterative model checking procedure for CFPs is already extended to pushdown processes, and this set is shown to be closed under parallel composition with finite processes.

The same holds here. All results of this paper carry over to the more general case. The tableau rules do not even have to be changed much, only the modality rules have to take care of a tuple of end states of nonatomic transitions. And the set of processes at each type level will also be closed under parallel composition with finite processes, in the same sense as pushdown processes are.

Several open questions concern the complexity of the methods. The k-exponential worst-case complexity is an upper bound, but I conjecture that it can also be established as a *lower* bound - for the worst case. In many cases, a clever algorithm

should perform much better. The tableau system gives the basis of an incremental algorithm, and the behavior of a program in the style of [34] would be interesting to observe. Very often, only a small subset of all parameter values is relevant, so one could save a lot.

Of more theoretical interest are two other questions. First, whether it is possible to give a *denotational* semantics to MPSs. And second (perhaps after solving the first problem), whether monadic second-order logic, which is more expressive than the mu-calculus, is decidable for macro processes. Also, of course, to extend the decision procedure to handle the full mu-calculus is something worth trying.

References

1. Andersen, H., *Model checking on boolean graphs.* ESOP '92, LNCS 582 (1992), 1–19.
2. Bergstra, J.A., and Klop, J.W., *Process theory based on bisimulation semantics.* LNCS 354 (eds de Bakker, de Roever, Rozenberg) (1988), 50–122.
3. Bradfield, J.C., *Verifying temporal properties of systems.* Birkhäuser, Boston (1992).
4. Bradfield, J.C., and Stirling, C. P., *Verifying temporal properties of processes.* Proc. CONCUR '90, LNCS 458 (1990), 115-125.
5. Burkart, O., and Steffen, B., *Model checking for context-free processes.* CONCUR '92, LNCS 630 (1992), 123–137.
6. Burkart, O., and Steffen, B., *Pushdown processes: Parallel composition and model checking.* Tech. Rep. Aachen/Passau (1994), 17 p. (1992), 123–137.
7. Caucal, D., and Monfort, R., *On the transition graphs of automata and languages.* WG 90, LNCS 484 (1990), 311–337.
8. Clarke, E.M., Emerson, E.A., and Sistla, A.P., *Automatic verification of finite state concurrent systems using temporal logic specifications.* ACM TOPLAS **8** (1986), 244–263.
9. Cleaveland, R., *Tableau-based model checking in the propositional mu-calculus.* Acta Inf. **27** (1990), 725-747.
10. Cleaveland, R., Parrow, J., and Steffen, B., *The concurrency workbench.* Workshop Automatic Verification Methods for Finite-State Systems, LNCS 407 (1989), 24–37.
11. Cleaveland, R., and Steffen, B., *Computing behavioral relations, logically.* ICALP '91, LNCS 510 (1991).
12. Cleaveland, R., and Steffen, B., *A linear-time model-checking algorithm for the alternation-free modal mu-calculus.* CAV 91, LNCS 575 (1992), 48–58.
13. Damm, W., *The IO- and OI-hierarchies,* TCS **20** (1982), 95–205.
14. Damm, W., and Goerdt, A., *An automata-theoretic characterization of the OI-hierarchy,* Inf. and Cont. 71 (1986), 1–32.
15. Emerson, E.A., and Lei, C.-L., *Efficient model checking in fragments of the propositional mu-calculus.* 1st LiCS (1986), 267–278.
16. Engelfriet, J., and Schmidt, E.M., *IO and OI,* JCSS **15** (1977), 328–353, and JCSS **16** (1978), 67–99.
17. Fischer, M.J., *Grammars with macro-like productions,* 9th Conf. Switching and Automata Theory, IEEE (1968), 131–142.
18. German, S. M., Clarke, E. M. and Halpern, J. Y., *Reasoning about procedures as parameters in the language L4,* Inf. and Comp. **83** (1989) 265–359. (Earlier version: 1st LiCS (1986) 11–25)
19. Goerdt, A., *A Hoare calculus for functions defined by recursion on higher types,* Logics of Programs 1985, LNCS 193, 106–117.

20. Habel, A., *Hyperedge replacement: Grammars and languages.* PhD thesis, Bremen (1989), 193 p.
21. Habel, A., and Kreowski, H.-J., *May we introduce to you: Hyperedge replacement,* Graph-grammars and their application to computer science 1986, LNCS 291 (1987), 15–26.
22. Hungar, H., *Complexity of proving program correctness,* TACS '91, LNCS 526 (1991), 459–474.
23. Hungar, H. *The complexity of verifying functional programs,* STACS '93, LNCS 665 (1993), 428–439.
24. Hungar, H., and Steffen, B., *Local model checking for context-free processes.* ICALP '93, LNCS 700 (1993), 593-605.
25. Huynh, D.T., and Tian, L., *Deciding bisimilarity of normed context-free processes is in* Σ_2^p. Tech. Rep. UTDCS-1-92, Univ. Texas Dallas (1992).
26. Iyer, S.P., *A note on model checking context-free processes,* North American Process Algebra Workshop '93 (ed. Bard Bloom).
27. Kowalczyk, W., Niwinski, D., and Tiuryn, J. *A generalization of Cook's auxiliary-pushdown-automata theorem,* Fund. Inf. **12** (1989) 497–506.
28. Kozen, D., *Results on the propositional* μ*-calculus.* TCS **27** (1983), 333–354.
29. Kfoury, A. J., Tiuryn, J. and Urzyczyn, P., *The hierarchie of finitely typed functions,* 2nd LiCS (1987) 225–235.
30. Larsen, K. G., *Proof systems for satisfiability in Hennessy-Milner logic with recursion.* TCS **72** (1990), 265-288.
31. Larsen, K.G., *Efficient local correctness checking.* CAV '92.
32. Muller, D.E., and Schupp, P.E., *The theory of ends, pushdown automata, and second-order logic.* TCS **37** (1985), 51–75.
33. Stirling, C. P., and Walker, D. J., *Local model checking in the modal mu-calculus.* TAPSOFT '89, LNCS 351 (1989), 369-383.
34. Vergauwen, B., and Lewi, J., *A linear local model checking algorithm for CTL.* CONCUR '93, LNCS 715 (1993), 447–461.
35. Winskel, G., *A note on model checking the modal mu-calculus.* ICALP '89, LNCS 372 (1989), 761–772.