Model Checking Value-passing Processes *

Huimin Lin
Laboratory for Computer Science
Institute of Software, Chinese Academy of Sciences
P.O Box 8718, Beijing 100080
Fax: +86 10 62563894
E-mail: lhm@ios.ac.cn

Key words: Model checking, mu-calculus, value-passing processes, verification algorithms.

Abstract

An algorithm for model checking value-passing processes is presented. Processes are modeled as symbolic transition graphs with assignments. *To specify properties for such processes a graphical predicate mu-calculus is introduced. It allows arbitrary nesting of the least and greatest fixpoints, and contains the propositional mu-calculus as a proper subset. The algorithm instantiates input variables on-the-fly and states are only generated when they are needed for the computation. To handle alternating fixpoints properly, a multi-stack is employed and the controlling strategy is such that a state is evaluated without depending on the default values for more deeply nested states. The algorithm is shown correct with respect to the semantics of the predicate mu-calculus. Its complexity is also analysed.*

1 Introduction

Research in model checking has so far focused on "ground" (*i.t.* variable-free) concurrent systems which can be modelled as *labelled transition systems*. Properties are specified as formulas in propositional modal logics, and algorithms can be invoked to automatically check if a system possesses the desired properties [2]. This framework is not convenient for applications which involve explicit manipulations of data variables, as in the case of value-passing processes.

Value-passing processes typically use input and output actions of the forms $c?x$ and $c!e$, respectively, to express inter-process communications. The process $c!e.P$ has the capacity of sending the value of expression e along channel c and then behave like P, while $c?x.Q$ can receive a message from channel c, assigning it to variable x which can then be used in the residual process Q. To specify temporal properties of such processes it is natural to introduce modalities corresponding to such actions as well as parameterised fixpoints. For example, a property we want to know about a communication protocol P is "It is always the case that whenever a value is received via channel r, after some internal activities it will eventually be sent out along channel s". In first-order μ-calculus this can be written succinctly as

$$F = \nu X[r?x]\mu Y(x/z)(\langle\tau\rangle Y(z) \vee \langle s!z\rangle X)$$

The formular uses the box modal operator $[r?x]$, to express "whenever an input action $r?x$ takes place", and the diamond operator $\langle s!z\rangle$, to express "an output action $s!z$ should happen". It also involves a parameterised fixpoint construction $\mu Y(x/z)(\langle\tau\rangle Y(z) \vee \langle s!z\rangle X)$ where x and z are the actual and formal parameters, respectively. Now our question can be formally posed as a model checking problem: does $P \models F$? Unfortunately, simple though, so far there is no automated verification tool can solve it for us.

In this paper we shall propose an algorithm which automatically decides whether a value-passing process possesses a property expressed in first-order μ-calculus, when the value domain involved is finite or "data-independent" (a value domain is data-independent if the only operations on its data values, which may be infinitely many, are tests for equality and inequality; a typical example of data-independent processes is communication protocols which do not perform functional calculations on the data being transmitted).

As may be guessed, a key issue in model checking value-passing processes is how to deal with input actions, since these are where data variables are introduced. The traditional approach is to translate them into so-called "basic processes", namely processes that do not involve data vari-

* Supported by the National Natural Science Foundation of China.

ables, by instantiating input variables with all possible values. The properties to check are then written in some standard propositional modal logics, and existing model checkers can be employed. This approach sounds appealing as it does not require any new mechanism. However, it has several disadvantages. For finite value domains the translation may result in exponential blow-up in the sizes of processes. Also, as structural information gets lost during the translation, the diagnosis messages generated by the model checkers do not point to the original presentation therefore are not very helpful. Furthermore, properties asking exact matching between values appearing at different places, such as the value sent on s and the value received on r in the formular F above, are cumbersome to express in propositional modal logics.

To overcome these drawbacks the algorithm proposed in this paper works directly on value-passing processes and first-order μ-calculus formulas, both represented as graphs. Processes are modeled as *symbolic transition graphs* [8]. Similarly we introduce a graphical predicate μ-calculus whose nodes correspond to logical/modal operators (such as $\wedge$, $\vee$, $[c?x]$, $\langle c?x\rangle$, *etc.* and edges denote the subterm relation (fixpoints give rise to cycles). Nodes are grouped into *blocks* each of which is associated with a *nesting depth* as well as a fixpoint-type which is either a μ (a least fixpoint) or a ν (a greatest fixpoint). Such modal graphs are used as a common representation for value-passing μ-calculi, because formulas in textual forms can be translated into them in a straightforward manner.

According to the semantics of the predicate μ-calculus, the algorithm asks for an input action from a process when and only when an input modality is encountered. In this case the input variables of the actions and of the modality are instantiated with the same elements of the value domain, one by one. For instance, suppose the process $c?x.P(x)$ is being checked against the formular $\langle c?y\rangle F(y)$ over the value domain $\{1, 2, \ldots, 10\}$. The on-the-fly instantiation strategy yields ten subsequent process-formular pairs $\langle P(i),\ F(i)\rangle$, $1 \leq i \leq 10$, to be further investigated. In contrast, the traditional "early instantiation" approach requires that the process and the formular be independently instantiated into $c_1.P_1 + c_2.P_2 + \ldots + c_{10}.P_{10}$ and $\langle c_1\rangle F_1 \vee \langle c_2\rangle F_2 \vee \ldots \langle c_{10}\rangle F_{10}$, respectively, before the matching starts. There are $10 \times 10 = 100$ comparisons to perform, with 90 definitely fail.

A direct consequence of the on-the-fly instantiation strategy is that search spaces can not be generated before model checking starts. Our algorithm creates states only when they are needed for the calculation of the final result. States are grouped into blocks according to their nesting depth. Since there exist mutual dependences between blocks (due to alternating fixpoints), a multi-stack consisting of a vector of stacks, one for each block, is employed. The algorithm performs a generalised depth-first search using the multi-stack. To handle alternating fixpoints properly, the controlling strategy ensures that no state will be evaluated under assumptions on the default values for states with greater nesting depths.

The rest of the paper is organised as follows. The model for value-passing processes used in this paper is reviewed in the next section. Section 3 introduces the graphical first-order μ-calculus. The model checking algorithm is presented in Section 4, together with correctness and complexity results. Section 5 addresses data independence. Finally the paper is concluded with Section 6 where related work is also discussed.

2. Symbolic Transition Graphs

Pure processes are modelled using *labelled transition systems* where transitions are labelled with synchronisation actions without internal structures. To model value-passing processes we extend labelled transition systems by associating each node a set of *data variables*. Accordingly, each transition is labelled with a triple $(b, \overline{x} := \overline{e}, \alpha)$ where b is a boolean expression giving the condition for the transition to be triggered, $\overline{x} := \overline{e}$ is multiple assignment to update the values of data variables when the transition takes place, and α is a communication action which is either an input, or an output, or a τ (representing internal computation). For the formal definition, let *Chan* be the set of channel names, *DVar* the set of data variables, *DExp* the set of data expressions, and *BExp* the set of boolean expressions. We write $fv(e)$ for the set of free variables of data/boolean expression e. Let also $Act = \{\, \tau, c?x, c!e \mid c \in Chan, x \in DVar, e \in DExp \,\}$ be the set of *communication actions*, arranged over by α. The sets of free/bound variables of communication actions are defined by $fv(c!e) = fv(e)$, $bv(c?x) = \{x\}$, and $fv(\alpha) = bv(\alpha) = \emptyset$ otherwise.

Definition 2.1 A symbolic transition graph with assignment (STGA for short) is a rooted directed graph where each node n is associated with a finite set of free variables $fv(n)$ and each edge is labelled by a triple $(b, \overline{x} := \overline{e}, \alpha)$. An STGA is *well-formed* if for each edge from n to m, written $n \overset{b,\overline{x}:=\overline{e},\alpha}{\longmapsto} m$, it holds that $fv(b, \overline{e}) \subseteq fv(n)$, $fv(\alpha) \subseteq \{\overline{x}\}$, and $fv(m) \subseteq \{\overline{x}\} \cup bv(\alpha)$. □

We will often simply write $n \overset{\theta,\alpha}{\longmapsto} m$ for $n \overset{true,\theta,\alpha}{\longmapsto} m$, and will also omit θ when it is the identity assignment on $fv(n)$.

Let *Eval* be the set of *evaluations* which are total mappings from *DVar* to *Val*. If $\rho \in Eval$ then $\rho\{x \mapsto v\}$ is the evaluation which differs from ρ only in that it maps x to v. This notion generalises to vectors in the expected way. If $\theta \equiv \overline{x} := \overline{e}$ then we define $\theta\rho = \rho\{\overline{x} \mapsto \rho(\overline{e})\}$. A STGA $\mathcal{G}$ induces a "partially ground" labelled transition system

$$\frac{m \overset{b,\overline{x}:=\overline{e},\tau}{\longmapsto} n}{m_\rho \overset{\tau}{\longrightarrow} n_{\theta\rho}} \quad \rho(b) = true$$

$$\frac{m \overset{b,\overline{x}:=\overline{e},c!e}{\longmapsto} n}{m_\rho \overset{c!\rho'(e)}{\longrightarrow} n_{\theta\rho}} \quad \rho(b) = true$$

$$\frac{m \overset{b,\overline{x}:=\overline{e},c?y}{\longmapsto} n}{m_\rho \overset{c?y}{\longrightarrow} n_{\theta\rho}} \quad \rho(b) = true$$

Figure 1. Operational Semantics

whose vertices are pairs of the form n_ρ where n is a node and ρ is an evaluation (supplying values for the free variables of n). Such pairs are called *processes*. Transitions are generated according to the operational semantics given in Figure 1. If $p \equiv n\rho$ we will write $p[v/x]$ for $n_{\rho\{x\mapsto v\}}$.

Transition systems so generated are "partially ground" because input actions are not yet instantiated. The instantiation will be carried out when graphs are model checked with respect to modal graphs to be introduced in the next section.

3 A Predicate μ-Calculus

In this section we introduce a graphical version of predicate μ-calculus and define its semantics. To demonstrate that it can be regarded as a general presentation for first-order μ-calculus, we will show how to translate the more familiar form of μ-calculus to the graphical form in Appendix A.

Let the set of *modal operators*, ranged over by β, be $ModOp = \{ [\alpha], \langle\alpha\rangle \mid \alpha \in Act \}$ and the sets of free and bound variables of a modal operator $fv([\alpha]) = fv(\langle\alpha\rangle) = fv(\alpha)$ and $bv([\alpha]) = bv(\langle\alpha\rangle) = bv(\alpha)$, respectively.

Definition 3.1 A *modal graph* is a directed graph $\mathcal{M} = (N, E, L_N, T, r)$ where N is the set of nodes, $E \subseteq N \times N$ the set of edges, $L_N = (L_V, L_O, L_D)$ associates each node $n \in N$ a set of free variable $L_V(n)$, an operator $L_O(n) \in \{\wedge, \vee, \forall x, \exists x, \beta, \theta\} \cup ModOp \cup BExp$, and a *depth* $L_D(n)$ which is a natural number. $r \in N$ is the *root* of $\mathcal{M}$. T assigns to each depth number a *fixpoint type* which is either μ or ν. We assume $\{ L_D(n) \mid n \in N \}$ is a set of consecutive numbers starting from 0 and $T(i) \neq T(i+1)$ for any $i \geq 0$.

A modal graph is subject to the following constrains:

- If $L_O(n) = b \in BExp$ then $fv(b) \subseteq L_V(n)$ and n has no outgoing edges.
- If $L_O(n) \in \{\forall x,\ \exists x\}$ then n has exactly one outgoing edge $n \longrightarrow n'$ and $fv(n') \subseteq fv(n) - \{x\}$.
- If $L_O(n) = \beta$ then n has exactly one outgoing edge $n \longrightarrow n'$ and $fv(n') \subseteq fv(n) \cup bv(\beta)$, $fv(\beta) \subseteq fv(n)$.
- If $L_O(n) = \theta \equiv \overline{x} := \overline{e}$ then n has exactly one outgoing edge $n \longrightarrow n'$ and $fv(\overline{e}) \subseteq fv(n)$, $fv(n') \subseteq \{\overline{x}\}$.
- If $L_O(n) \in \{\wedge,\ \vee\}$ then n has two outgoing edges, and $fv(n') \subseteq fv(n)$ for each $n \longrightarrow n'$. □

We will use the following abbreviations for a node n:

b	if $L_0(n) = b \in BExp$
$\theta n'$	if $L_0(n) = \theta$ and $n \longrightarrow n'$
$n_1 \wedge n_2$	if $L_0(n) = \wedge$ and $n \longrightarrow n_1,\ n \longrightarrow n_2$
$n_1 \vee n_2$	if $L_0(n) = \vee$ and $n \longrightarrow n_1,\ n \longrightarrow n_2$
$\forall x n'$	if $L_0(n) = \forall x$ and $n \longrightarrow n'$
$\exists x n'$	if $L_0(n) = \exists x$ and $n \longrightarrow n'$
$\langle c!e\rangle n'$	if $L_0(n) = \langle c!e\rangle$ and $n \longrightarrow n'$
$[c!e]n'$	if $L_0(n) = [c!e]$ and $n \longrightarrow n'$
$\langle c?x\rangle n'$	if $L_0(n) = \langle c?x\rangle$ and $n \longrightarrow n'$
$[c?x]n'$	if $L_0(n) = [c?x]$ and $n \longrightarrow n'$

The semantics of a modal graph $\mathcal{M}$ is defined with respect to a labelled transition system (induced from a STGA) which we assume to be fixed. To give meaning to $\mathcal{M}$ we first assign meaning to each node. Let S be the set of states of the labelled transition system and $\Gamma = \{ \zeta \mid \zeta : N \longrightarrow Eval \longrightarrow 2^S \}$. The set inclusion ordering on 2^S can be generalised to Γ by letting $\zeta \leq \zeta'$ if and only if $\zeta n\rho \subseteq \zeta' n\rho$ for all n and ρ.

The semantic function $\|\cdot\| : N \longrightarrow Eval \longrightarrow \Gamma \longrightarrow 2^S$ is defined according to the following rules:

$$\|b\|\rho\zeta = \begin{cases} S & \text{if } \rho(b) = true \\ \emptyset & \text{otherwise} \end{cases}$$

$$\|\theta n'\|\rho\zeta = \zeta n'(\theta\rho)$$

$$\|n_1 \wedge n_2\|\rho\zeta = \zeta n_1\rho \cap \zeta n_2\rho$$

$$\|n_1 \vee n_2\|\rho\zeta = \zeta n_1\rho \cup \zeta n_2\rho$$

$$\|\forall x n'\|\rho\zeta = \textstyle\bigcap_{v\in Val} \zeta n'\rho\{x \mapsto v\}$$

$$\|\exists x n'\|\rho\zeta = \textstyle\bigcup_{v\in Val} \zeta n'\rho\{x \mapsto v\}$$

$$\|\langle a\rangle n'\|\rho\zeta = \{ s \mid \exists s'\ s \overset{a}{\longrightarrow} s' \text{ and } s' \in \zeta n'\rho \}$$

$$\|[a]n'\|\rho\zeta = \{ s \mid \forall s'\ s \overset{a}{\longrightarrow} s' \text{ implies } s' \in \zeta n'\rho \}$$

$$\|\langle c!e\rangle n'\|\rho\zeta = \{ s \mid \exists s'\ s \overset{c!\rho(e)}{\longrightarrow} s' \text{ and } s' \in \zeta n'\rho \}$$

$$\|[c!e]n'\|\rho\zeta = \{ s \mid \forall s'\ s \overset{c!\rho(e)}{\longrightarrow} s' \text{ implies } s' \in \zeta n'\rho \}$$

$$\|\langle c?x\rangle n'\|\rho\zeta = \{ s \mid \begin{array}{l} \exists s'\ s \overset{c!y}{\longrightarrow} s' \text{ and } \forall v \in Val \\ s'[v/y] \in \zeta n'\rho\{x \mapsto v\} \end{array} \}$$

$$\|[c?x]n'\|\rho\zeta = \{ s \mid \begin{array}{l} \forall s'\ s \overset{c?y}{\longrightarrow} s' \text{ implies } \forall v \in Val \\ s'[v/y] \in \zeta n'\rho\{x \mapsto v\} \end{array} \}$$

There are some possible variations for the semantics of the input modalities. Instead of asking the residual process to satisfy the target proposition for *every* value in the data domain ("tight"), as in the above definitions, one could require this only for *some* value ("loose"). Also the order between matching input actions and instantiating input variables may be reversed, yielding a semantics in which input variables are instantiated before the comparison between input actions is performed. Such semantics is usually referred to as "early", in contrast to the "late" one as given above. Another variation concerns the diamond output modality: $\langle c!x \rangle$ could be treated as a binder [4, 11]. It is straightforward to modify the semantic definition and the model checking algorithm (in next section) to cater for these variations. Which semantics to use depends on intended applications.

Given a modal graph $\mathcal{M}$, let its ith *block* B_i be the set $\{ n \in N \mid L_D(n) = i \}$. If $j < i$ then B_j is called an *outer block* of B_i and B_i an *inner block* of B_j. Assume $\mathcal{M}$ have $k+1$ blocks $\{ B_i \mid 0 \leq i \leq k \}$, with $T(i) = \sigma_i \in \{\mu, \nu\}$. Let $\mathcal{B}_i = \{ B_j \mid i \leq j \leq k \}$, $0 \leq i \leq k$. To give semantics to $\mathcal{M}$, we assign meaning to each $\mathcal{B}_i$ by induction on i. To this end, with respect to an environment ζ, we associate a function $\mathbf{F}_{\zeta,i} : \Gamma \longrightarrow \Gamma$ to $\mathcal{B}_i$. As the base of induction let $\mathbf{F}_{\zeta,k+1}(\eta) = \eta$ for any η. Now assume $i \leq k$. Then $\mathcal{B}_i = \{B_i\} \cup \mathcal{B}_{i+1}$, and $\mathbf{F}_{\zeta,i}$ is defined by:

$$\mathbf{F}_{\zeta,i}(\eta)n\rho = \begin{cases} \|n\|\rho([\![\mathcal{B}_{i+1}]\!]\zeta\{\eta\lceil_{B_i}\}) & n \in B_i \\ ([\![\mathcal{B}_{i+1}]\!]\zeta\{\eta\lceil_{B_i}\})n\rho & \text{otherwise} \end{cases}$$

where $\eta\lceil_{B_i}$ is the restriction of η to B_i and $\zeta\{\eta\lceil_{B_i}\}$ is the modification to ζ by $\eta\lceil_{B_i}$ defined thus: $(\zeta\{\eta\lceil_{B_i}\})n = \begin{cases} \eta n & n \in B_i \\ \zeta n & \text{otherwise} \end{cases}$. Now let $[\![\mathcal{B}_i]\!]$ be the least fixpoint of $\mathbf{F}_{\zeta,i}$ if $\sigma_i = \mu$, and the greatest fixpoint of $\mathbf{F}_{\zeta,i}$ if $\sigma_i = \nu$.

In the above definition of $\mathbf{F}_{\zeta,i}$, $[\![\mathcal{B}_{i+1}]\!]\zeta\{\eta\lceil_{B_i}\}$ represents the meaning of $\mathcal{B}_{i+1}$ in an environment in which the nodes in B_i are interpreted according to η and the other nodes not in $\mathcal{B}_{i+1}$ (that is, those nodes in the outer blocks of B_i) are interpreted according to ζ. Since B_0 has no outer blocks, $[\![\mathcal{B}_0]\!]\zeta$ does not rely on ζ, therefore $[\![\mathcal{B}_0]\!]$ can be regarded as a function from *Eval* to $2^{\mathcal{S}}$. We take $[\![\mathcal{B}_0]\!]$ as the meaning of the graph $\mathcal{M}$.

We will need the notion of *alternation depth* of a modal graph when discussing the complexity of model checking algorithms. Since a model graph is a directed graph, it can be partitioned into maximal strongly connected components. Let the alternation depth of a maximal strongly connected component *SCC* be $1 + \max\{ L_D(n) - L_D(m) \mid n, m \in SCC \}$. The alternation depth of a modal graph is then defined as the maximum of the alternation depths of all its *SCC*s.

4 The Algorithm

Given a value-passing process p represented by q_{ρ_0} where q is a node of a STGA $\mathcal{G}$, and a proposition f represented by $r\rho'_0$ where r is the root of a modal graph $\mathcal{M}$, the model checking problem is to determine if $p \models f$, *i.t.* if $p \in [\![r\rho'_0]\!]$. In this section we assume $\mathcal{G}$ and $\mathcal{M}$ are over a finite data domain *Val*.

The model checking algorithm is presented in Figure 2. It belongs to the family of "local" algorithms ([6, 1, 12, 3]) as it only visits the part of search space needed for the computation. States are generated in a demand-driven fashion. In particular input variables are instantiated "on-the-fly", when input modalities are checked. A state s in the underlying search space $\mathcal{S}$ is a pair consisting of a process and a proposition, which takes the form $(m\rho,\ n\rho')$ where m and n are nodes of $\mathcal{G}$ and $\mathcal{M}$, respectively. We shall talk about "nesting depth", "fixpoint type", *etc* of s, referring to those of n. An edge from $(m_1\rho_1,\ n_1\rho'_1)$ to $(m_2\rho_2,\ n_2\rho'_2)$ consists of a pair of edges, $m_1\rho_1 \longrightarrow m_2\rho_2$ in the labelled transition system induced by $\mathcal{G}$ and $n_1 \longrightarrow n_2$ in $\mathcal{M}$. The notion of alternation depth of $\mathcal{S}$ is defined in the same way as that of a modal graph. Since the projection of a strongly connected component of $\mathcal{S}$ on the second component of its states is a strongly connected component of $\mathcal{M}$, the alternation depth of $\mathcal{S}$ can not exceed that of $\mathcal{M}$.

The algorithm employs a variant of "depth first search" in the underlying state space. Standard depth first search procedure has a very simple controlling mechanism: a single stack. However, this is not adequate for our problem because states are partitioned into blocks according to their nesting depths, and there are mutual dependences between different blocks due to alternating fixpoints. This suggests to use a multi-stack consisting of an array of stacks, one for each block. States in the same block will be pushed onto/popped from the same stack observing the last-in-first-out discipline, while between blocks the deeper nested ones have higher priority. To be more precise, let us arrange the blocks and stacks according to their nesting depths. A state of depth i is pushed onto the i-th stack. But in the main loop of depth first search it is always the state on the top of the last non-empty stack gets examined, even though it might not be the last state pushed onto the multi-stack. Let us call such a state the "top element" of the multi-stack.

Each state s has five attributes: $s.depth$, $s.status$, $s.\sigma$, $s.D$ and $s.instack$. $s.depth$ is the nesting depth of s. The $s.status$ attribute reflects the current status of s: `FRESH` indicates s is newly created, `VISITED`(b) means s has been encountered before with boolean value b. The $s.\sigma$ attribute is a constant used to initialize $s.status$ when it is created; it is *true* if s is a ν state and *false* otherwise. $s.D$ is the set of states whose values directly depend on the current value of s, therefore when the value of s is changed the values of the states in $s.D$ must be recomputed. Finally $s.instack$ is a boolean indicating if s is currently in the multi-stack. It is set to *true* when s is pushed onto the stack and to *false* when it is popped.

Initially the multi-stack contains only the target state, the

$\texttt{stack}[0\ \ldots\ (k{-}1)]$
$\texttt{pushStack}(s)$ = push s onto $\texttt{stack}[s.\mathit{depth}]$
$\texttt{top}()$ = return the top element of the first non-empty stack with the deepest depth
$\texttt{pop}()$ = remove from the stack the element returned by the last call to $\texttt{top}$

$\texttt{modelCheck}(s)$ = $\{$ $s.status := \texttt{VISITED}(s.\sigma)$; push$(s)$;
  while $\texttt{stack}$ is non-empty do $\texttt{close}(\texttt{top}())$;
  return $s.status$ $\}$

$\texttt{close}(s$ as $(p, n\rho))$ =
let $\texttt{checkAnd}(W)$ =
  let $(B_i, D_i) = \texttt{check}(s_i)$ for each $s_i \in W$
  in if there is some $B_i = \texttt{DEFERRED}$ then return($\texttt{DEFERRED}$)
    else if there is some $B_i = \texttt{VALUE}(\mathit{false})$ then $\{$add s to D_i; return($\texttt{VALUE}(\mathit{false})$)$\}$
    else $\{$add s to each D_i; return($\texttt{VALUE}(\mathit{true})$)$\}$
  $\texttt{checkOr}(W)$ =
  let $(B_i, D_i) = \texttt{check}(s_i)$ for each $s_i \in W$
  in if every $B_i = \texttt{DEFERRED}$ then return($\texttt{DEFERRED}$)
    else if there is some $B_i = \texttt{VALUE}(\mathit{true})$ then $\{$add s to D_i; return($\texttt{VALUE}(\mathit{true})$)$\}$
    else $\{$add s to each D_i; return($\texttt{VALUE}(\mathit{false})$)$\}$
  $B =$ case $L_O(n)$ of
    $\mathit{be} \Longrightarrow \rho(\mathit{be})$
    $\mid\ \overline{x} := \overline{e} \Longrightarrow \texttt{checkAnd}(\{\, (p, (n',\ \rho\{\overline{x} \mapsto \rho(\overline{e})\})) \mid n \longrightarrow n' \,\})$
    $\mid\ \wedge \Longrightarrow \texttt{checkAnd}(\{\, (p, (n_i,\ \rho)) \mid n \longrightarrow n_i \,\})$
    $\mid\ \vee \Longrightarrow \texttt{checkOr}(\{\, (p, (n_i,\ \rho)) \mid n \longrightarrow n_i \,\})$
    $\mid\ [c!e] \Longrightarrow \texttt{checkAnd}(\{\, (p_i, (n',\ \rho)) \mid p \xrightarrow{c!\rho(e)} p_i,\ n \longrightarrow n' \,\})$
    $\mid\ \langle c!e \rangle \Longrightarrow \texttt{checkOr}(\{\, (p_i, (n',\ \rho)) \mid p \xrightarrow{c!\rho(e)} p_i,\ n \longrightarrow n' \,\})$
    $\mid\ [c?x] \Longrightarrow \texttt{checkAnd}(\{\, (p_i[v/y], (n', \rho\{x \mapsto v\})) \mid n \longrightarrow n', p \xrightarrow{c?y} p_i, v \in \mathit{Val} \,\})$
    $\mid\ \langle c?x \rangle \Longrightarrow \texttt{checkOr}(\{\, (p_i[v/y], (n', \rho\{x \mapsto v\})) \mid n \longrightarrow n', p \xrightarrow{c?y} p_i, v \in \mathit{Val} \,\})$
    $\mid\ \forall x \Longrightarrow \texttt{checkAnd}(\{\, (p, (n', \rho\{x \mapsto v\})) \mid n \longrightarrow n', v \in \mathit{Val} \,\})$
    $\mid\ \exists x \Longrightarrow \texttt{checkOr}(\{\, (p, (n', \rho\{x \mapsto v\})) \mid n \longrightarrow n', v \in \mathit{Val} \,\})$
in if $B = \texttt{VALUE}(b)$ then
  $\{$pop(); if $s.status = \texttt{VISITED}(b')$ and $b' \neq b$ then
    $\{s.status := \texttt{VISITED}(b)$; $\texttt{restore}(s.D, b)\}\}$

$\texttt{check}(s)$ = case $s.status$ of
  $\texttt{FRESH} \Longrightarrow \{s.status := \texttt{VISITED}(s.\sigma)$; $\texttt{push}(s)$; return($\texttt{DEFERRED}$, $s.D$)$\}$
  $\mid$ $\texttt{VISITED}$(b) $\Longrightarrow$ return($\texttt{VALUE}(b)$, $s.D$)

$\texttt{restore}(D, b)$ = for each $s \in D$
  if $s.status = \texttt{VISITED}(b')$ and $b' \neq b$ then
    $\{$ $s.status := \texttt{VISITED}(s.\sigma)$;
    if $\neg s.instack$ then push(s);
    $\texttt{restore}(s.D, b)$;
    $s.D := \emptyset$ $\}$

Figure 2. The algorithm

process-proposition pair to be model checked. The main function `modelCheck` consists of a simple while-loop. Each loop examines the current top element of the multi-stack by applying function `close` to it. When the entire multi-stack is empty the while-loop terminates with the *status* attribute of the target state holding the result (either "*true*" or "*false*").

The function `close` tries to compute a value for its argument s based on the fixpoint type of s and the current values of its successors. Due to the existence of cycles, some of the successors may have been explored before and thus have values computed, some may not. If there are not enough successors with computed values to determine the value of the current state then one of its successors is pushed onto the stack and the computation of the current state is deferred. The state remains in the stack and will be processed again later, when more information on its successors becomes available. Otherwise the value for s is calculated using function `checkAnd` or `checkOr`, depending on s's operator type, and the state is removed from the stack. Besides a truth value `checkAnd`/`checkOr` also adds s to $s_i.D$ for each of its successor state s_i on which the returned truth value depends. When the calculation is completed we compare s's new value b with its previous value (if any). In case they are different then all the states whose values depend on the previous value of s must be reprocessed. This is accomplished by function `restore` which recursively pushes every state depending on s onto the multi-stack. Because of monotonicity not all the states depending on s need reprocessing – only those whose values are different from b do.

Note that because of the way the multi-stack is organised, a state s gets evaluated only when all the states in its inner blocks whose values are needed for the calculation have been evaluated. In other words, the evaluation of a state may depend on the default values of the states in its outer blocks, but not on those in its inner blocks.

Correctness

Let $\mathcal{R}$ be the search space that has been explored during the execution of the algorithm. States in $\mathcal{R}$ are partitioned into subsets $\{\, C_i \mid 0 \leq i \leq k \,\}$ (some C_i:s may be empty), also called *blocks*, according to their nesting depths. Let $C'_i = \{\, s \in C_i \mid s \text{ is not in the stack} \,\}$ and $\mathcal{R}' = \bigcup C'_i$. The current content of $\mathcal{R}$ can be regarded as a function $\zeta_{\mathcal{R}} : N \longrightarrow \mathit{Eval} \longrightarrow 2^{\mathcal{S}}$ by letting

$$\zeta_{\mathcal{R}} n\rho = \{\, p \mid \begin{array}{l} \exists s \in \mathcal{R} \text{ s.t. } s = (p, n\rho) \text{ and} \\ s.status = \mathtt{VISITED}(\mathit{true}) \end{array} \,\}$$

Using the notation introduced in Section 3 let $[\![\mathcal{C}_i]\!]\zeta_{\mathcal{R}}$ denote the meaning of $\mathcal{C}_i = \{\, C_j \mid i \leq j \leq k \,\}$, $0 \leq i \leq k$, with respect to $\zeta_{\mathcal{R}}$.

The following proposition asserts that the values computed by function *close*, for states in the current search space but not in the stack, are consistent with the semantic definition. It is the key step towards the correctness result.

Proposition 4.1 *The while loop in* `modelCheck` *has the following invariant INV:*

$$\textit{for any } s = (p, n\rho) \in \mathcal{R}', s.status = \mathtt{VISITED}(\mathit{true}) \\ \textit{iff } p \in ([\![\mathcal{C}_0]\!]\zeta_{\mathcal{R}})n\rho.$$

Proof: (Sketch) Initially $\mathcal{R}'$ is empty and the invariant holds trivially. Now we show `close` maintains the invariant. Suppose *INV* holds when `close` is called and $\mathtt{top}() = s$ with $s.status = \mathtt{VISITED}(b')$. We perform a case analysis on the value B calculated in the body of `close`. If $B = \mathtt{DEFERRED}$ then some new state with default value is pushed onto the stack, s remains in the stack and $\mathcal{R}'$ is unchanged, hence *INV* still holds. Otherwise let $B = \mathtt{VISITED}(b)$. In this case a new value is calculated for s and s is moved from the stack to $\mathcal{R}'$. If $b = b'$ then nothing else is changed. Since the calculation of B follows the definition of the semantic function $||\cdot||\rho\zeta_{\mathcal{R}}$, and the evaluation of a state does not depend on the default values of any states in its inner blocks, *INV* is not violated. If $b \neq b'$, then all the states whose values depend on the previous value of s are moved from $\mathcal{R}'$ to the stack, therefore *INV* remains hold. □

Theorem 4.2 *Upon termination, the algorithm correctly computes* $[\![C_0]\!]\zeta$ *for any* ζ.

Proof: When the algorithm terminates, the stack is empty and $R = R'$, *i.e.* the search space is closed. Therefore the value of $[\![C_0]\!]\zeta_{\mathcal{R}}$ does not depend on $\zeta_{\mathcal{R}}$. Hence, according to Proposition 4.1, R coincides with $[\![C_0]\!]\zeta$ for any ζ. □

Complexity

Let $\hat{\mathcal{G}}$ be the subgraph of the labelled transition system induced by $\mathcal{G}$ reachable from p. Write $|\, \hat{\mathcal{G}} \,|$ and $|\, \longrightarrow_{\hat{\mathcal{G}}} \,|$ for the number of nodes and the number of edges of $\hat{\mathcal{G}}$, respectively. Write also $|\, \mathcal{M} \,|$ for the number of nodes of $\mathcal{M}$, $|\, \mathit{Val} \,|$ for the size of the value domain and V for $max\{\, |\, fv(n) \,| \mid n \text{ is a node of } \mathcal{M} \,\}$. Then the number of states of the search space, $|\, \mathcal{S} \,|$, is bounded by $|\, \hat{\mathcal{G}} \,| \cdot |\, \mathcal{M} \,| \cdot |\, \mathit{Val} \,|^V$ and the number of edges, $|\, \longrightarrow_{\mathcal{S}} \,|$, is bounded by $(|\, \longrightarrow_{\hat{\mathcal{G}}} \,| + 2|\, \hat{\mathcal{G}} \,|) \cdot |\, \mathcal{M} \,| \cdot |\, \mathit{Val} \,|^V$.

Theorem 4.3 *The algorithm executes in time bounded by* $c|\, \mathcal{S} \,|^{k+1} \cdot \log |\, \mathcal{S} \,|$, *where* c *is a constant and* k *the alteration depth of the modal graph, and in space linear to the size of* $\mathcal{S}$.

Proof: (Sketch) Consider a *SCC* of $\mathcal{S}$. It can be decomposed into blocks $C_i,\ C_{i+1}, \ldots,\ C_j$ according to nesting depths. Since the evaluation of a state in the outer most block C_i does not depend on the default values of states in its inner blocks, it can change its value at most once. Similarly, a state in C_{i+1} can change its value at most $\mid C_i \mid$ times, ... Therefore the number of calls of `close` for evaluating the *SCC* is bounded by $\sum_{l=i}^{j} \mid C_i \mid \cdot \mid C_{i+1} \mid \ldots \mid C_l \mid \ \leq\ c \mid \cup_{i \leq l \leq j} C_l \mid^{j-i+1}$. The total number of calls is the sum of those taken for all the *SCC*s of $\mathcal{S}$, which is bounded by $c \mid \mathcal{S} \mid^k$. The `restore` operation takes time proportional to $\mid \mathcal{S} \mid$. The log fact is incurred by accessing states, under the assumption that the search space is organised as a balanced binary tree.

For the space complexity, first note that the size of the multi-stack can not exceed $\mid \mathcal{S} \mid$. Moreover, for each state s, the size of $s.D$ can not exceed the number of edges leading to s, hence the total space required to store the direct dependency relation is bounded by $\mid \longrightarrow_{\mathcal{S}} \mid$. The size of the other data structures used by the algorithm is clearly bounded by the number of states. Therefore the overall space consumed by the algorithm is linear to the size of the (explored) search space. □

5 Data Independence

In the previous section data variables are assumed to range over finite value domains. This restriction can be relaxed to allow the so-called "data independent" domains [13, 5, 7]. A data independent domain may be infinite, but has only a finite number of constant symbols; variables and constants of a data independent domain may be tested for equality or inequality, but they are not subject to other operations nor first-order quantifications.

It is not difficult to see that, to allow data independent domains, only the part dealing with input modalities of the algorithm needs modifying. Let *Data* be such a domain with a finite set of constants C. We use a countable set of "symbolic values" which are different from these constants and are totally ordered. The function $n\mathit{Val}(p, n\rho)$, where p has the form $m\rho'$, returns the least symbolic value that does not appear in $\mathit{cod}(\rho'|_{fv(m)}) \cup \mathit{cod}(\rho|_{fv(n)})$. Intuitively, a data independent variable should be instantiated with all constants (to resolve subsequent equality/inequality tests between this variable and constants), plus a new symbolic value (to resolve subsequent comparisons between this variable and other variables). The modified version of the input modality part of the algorithm is as follows:

$$
\begin{aligned}
B = \ & \text{case } L_O(n) \text{ of} \\
& \quad \ldots \\
& \mid \ [c?x] \Rightarrow \texttt{checkAnd}(\{(p_i[v/y], (n', \rho\{x \mapsto v\})) \mid \\
& \qquad\qquad \{n \longrightarrow n', p \xrightarrow{c?y} p_i, v \in U\})
\end{aligned}
$$

$$
\text{where } U = \begin{cases} C \cup n\mathit{Val}(p, n\rho) & \text{if } x \text{ is over } \mathit{Data} \\ \mathit{Val} & \text{otherwise} \end{cases}
$$

6 Conclusion and Related Work

We have proposed an approach to model checking value-passing processes. It consists of a graphical predicate μ-calculus and an algorithm operating on modal graphs. The graphical logic is intended to serve as a common representation for value-passing modal logics. The algorithm is *truly local* in the sense that it does not require a prior construction of the search space. This is achieved by instantiating input variables and generating states "on-the-fly", a strategy inherited from our previous work on computing bisimulations for value passing processes [9]. The algorithm has been implemented in Standard ML, as part of our efforts to build an automated verification tool for value-passing systems.

The algorithm of Figure 2 has a time complexity which is in power $k + 1$ of the size of the search space. We have a refined version which is in power k. It uses a counter for each state, to remember which of its successors have been processed, as well as two numbers to record how many successors evaluated to *true* and *false*, respectively. As the refined algorithm involves so many details that it is difficult to present it in a short paper, we have chosen to include the "slower" but simpler version in this paper.

Currently we are investigating how to generate informative diagnosis messages which will be useful in debugging a system when the answer returned by the algorithm is "No".

There is a sea of publications on model checking, but only a few address value-passing processes. A modal logic for value-passing processes was first proposed in [4], and later extended with fixpoints in [11]. The main concern of these two papers is to formulate proof systems for deriving statements asserting a process satisfies a formular. Algorithms for model checking value-passing processes against such modal logics, which is the main subject of the present paper, was not considered in either paper.

Recently [10] proposes to encode model checking problems for value-passing processes as prolog programs. The property logic includes input and output modalities as well as non-parameterised fixpoints. Value-passing processes are expanded into ground transition systems, by instantiating input variables with all values (as a consequence data independent processes can not be handled). Formulas are encoded as prolog clauses. Model checking is carried out by executing these clauses on the resulting transition systems.

Acknowledgments Thanks to Xinxin Liu for carefully reading a draft of this paper and suggesting several improvements.

References

[1] H. Andersen. Model checking and boolean graphs. *Theoretical Computer Science*, 126:3–30, 1994.

[2] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.

[3] R. Cleaveland. Tableau-based model checking in the propositional mu-calculus. *Acta Informatica*, 27(4):724–747, 1990.

[4] M. Hennessy and X. Liu. A modal logic for message passing processes. *Acta Informatica*, 32:375–393, 1995.

[5] B. Jonsson and J. Parrow. Deciding bisimulation equivalences for a class of non-finite-state programs. *Information and Computation*, 107:272–302, 1993.

[6] K. Larsen. Efficient local correctness checking. In *CAV'92*, volume 663 of *Lecture Notes in Computer Science*, pages 30–43. Springer–Verlag, 1992.

[7] R. Lazic and D. Nowak. A unifying approach to data-independence. In *CONCUR2000*, Lecture Notes in Computer Science. Springer–Verlag, 2000.

[8] H. Lin. Symbolic transition graph with assignment. In *CONCUR'96*, volume 1119 of *Lecture Notes in Computer Science*, pages 50 – 65. Springer–Verlag, 1996.

[9] H. Lin. "On-the-fly instantiation" of value-passing processes. In *FORTE/PSTV'98*, pages 215–230. Kluwer Academic Publishers, 1998.

[10] C. Ramakrishnan. A model checker for value-passing mu-calculus using logic programming. In *3rd Int'l Symposium on Practical Aspects of Declarative Languages*, March, 2001.

[11] J. Rathke and M. Hennessy. Local model checking for value-passing processes. *Proceeding of the 9th International Conference on Theoretical Aspects of Computer Software*, 1997.

[12] C. Stirling and D. Walker. Local model checking in the modal mu-calculus. *Theoretical Computer Science*, 89:161–177, 1991.

[13] P. Wolper. Expressing interesting properties of programs in propositional temporal logic (extended abstract). In *Proc. 13th ACM POPL*, pages 184–193, January 1986.

A From Textual μ-Calculus to Modal Graphs

Let p, q range over atomic propositions. We use predicate variables, ranged over by X, Y *etc*, to form recursive predicates. Each predicate variable has associated with it an *arity* which is a non-negative integer number. The syntax of the first-order μ-calculus can then be defined by the following BNF grammar where β ranges over *ModOp*:

$$\begin{array}{lcl} \phi & ::= & p \mid \phi \wedge \phi \mid \phi \vee \phi \mid \forall x \phi \mid \exists x \phi \mid \beta \phi \mid \Lambda \overline{e} \\ \Lambda & ::= & (\overline{x})\phi \mid X \mid \mu X \Lambda \mid \nu X \Lambda \end{array}$$

As can be seen the language is divided into two syntactical categories, *propositions* (ϕ) and *predicates* (Λ). Predicates are functions from data values to propositions. The x in $\forall x\phi$, $\exists x\phi$, $[c?x]\phi$ and $\langle c?x\rangle\phi$ are binding occurrence with scope ϕ. The $\overline{x}$ in $(\overline{x})\phi$ is a vector of distinct data variables which are also binding with scope ϕ. These induce the usual notions of free and bound data variables for propositions, denoted by $fv(\phi)$ and $bv(\phi)$, respectively. We shall only use data-closed predicates to form recursion so in $(\overline{x})\phi$ it is always required $fv(\phi) \subseteq \{\overline{x}\}$. As a consequence we have $fv(\Lambda\overline{e}) = fv(\overline{e})$. The arity of $(\overline{x})\phi$ is equal to the length of $\overline{x}$ and we require arities be respected when forming abstractions $\Lambda\overline{e}$, $\mu X \Lambda$ and $\nu X \Lambda$. For the sake of readability we sometimes write $\mu X(\overline{e}/\overline{x})\phi$ for $(\mu X(\overline{x})\phi)\overline{e}$, as in the protocol example discussed in the Introduction.

Instead of defining semantics for this logic, we give a translation from it to the graphical modal logic of Section 3. Let ϕ_0 be a proposition. By α-conversion we may assume that all bound variables in ϕ_0 are different. The set of nodes of the modal graph is $N = \{ \phi \mid \phi \text{ is a subformular of } \phi_0 \}$. The set of free variables of node ϕ is just $fv(\phi)$. The edges and labeling functions are defined recursively using a family of mappings $\mathcal{M}_i^\sigma$, parameterised by integer i and $\sigma \in \{\mu, \nu\}$, as follows (the dual cases for $\vee$, $\exists$ and ν are omitted):

$$\mathcal{M}_i^\sigma(\beta\phi) \longrightarrow \mathcal{M}_i^\sigma(\phi) \qquad \begin{array}{l} L_O(\beta\phi) = \beta \\ L_D(\beta\phi) = i \\ T(i) = \sigma \end{array}$$

$$\mathcal{M}_i^\sigma(\phi \wedge \psi) \begin{array}{l} \longrightarrow \mathcal{M}_i^\sigma(\phi) \\ \longrightarrow \mathcal{M}_i^\sigma(\psi) \end{array} \qquad \begin{array}{l} L_O(\phi \wedge \psi) = \wedge \\ L_D(\phi \wedge \psi) = i \\ T(i) = \sigma \end{array}$$

$$\mathcal{M}_i^\sigma(\forall\phi) \longrightarrow \mathcal{M}_i^\sigma(\phi) \qquad \begin{array}{l} L_O(\forall\phi) = \forall \\ L_D(\forall\phi) = i \\ T(i) = \sigma \end{array}$$

$$\begin{array}{l} \mathcal{M}_i^\mu((\mu X(\overline{x})\phi)\overline{e}) \longrightarrow \\ \quad \mathcal{M}_i^\mu(\phi[\mu X(\overline{x})\phi/X]) \end{array} \qquad \begin{array}{l} L_O((\mu X(\overline{x})\phi)\overline{e}) = \overline{x} := \overline{e} \\ L_D((\mu X(\overline{x})\phi)\overline{e}) = i \\ T(i) = \mu \end{array}$$

$$\begin{array}{l} \mathcal{M}_i^\mu((\nu X(\overline{x})\phi)\overline{e}) \longrightarrow \\ \quad \mathcal{M}_{i+1}^\nu(\phi[\nu X(\overline{x})\phi/X]) \end{array} \qquad \begin{array}{l} L_O((\nu X(\overline{x})\phi)\overline{e}) = \overline{x} := \overline{e} \\ L_D((\nu X(\overline{x})\phi)\overline{e}) = i+1 \\ T(i) = \nu \end{array}$$

Given a proposition ϕ we let its graphical form be $\mathcal{M}_0^\sigma(\phi)$ where σ is the top-most operator of ϕ if it is μ or ν, or μ otherwise. It is not difficult to see that the set of nodes of the graph equals to the set of subformulas of ϕ, therefore is finite.