A Model Checker for Value-Passing Mu-Calculus using Logic Programming*

C. R. Ramakrishnan

Department of Computer Science,
SUNY at Stony Brook
Stony Brook, NY 11794–4400, USA
E-mail: cram@cs.sunysb.edu

Abstract. Recent advances in logic programming have been successfully used to build practical verification toolsets, as evidenced by the XMC system. Thus far, XMC has supported value-passing process languages, but has been limited to using the propositional fragment of modal mu-calculus as the property specification logic. In this paper, we explore the use of data variables in the property logic. In particular, we present value-passing modal mu-calculus, its formal semantics and describe a natural implementation of this semantics as a logic program. Since logic programs naturally deal with variables and substitutions, such an implementation need not pay any additional price— either in terms of performance, or in complexity of implementation— for having the added flexibility of data variables in the property logic. Our preliminary implementation supports this expectation.

1 Introduction

XMC is a toolset for specifying and verifying concurrent systems [RRS+00]. Verification in XMC is based on temporal-logic model checking [CES86]. In its current form, temporal properties are specified in the alternation-free fragment of the modal mu-calculus [Koz83]; and system models are specified in XL, a process language with data variables and values, based on Milner's CCS [Mil89]. The computational components of the XMC system, namely, the compiler for the specification language, the model checker, and the evidence generator are built on top of the XSB tabled logic programming system [XSB].

XMC started out in late 1996 as a model checker for basic CCS— i.e., CCS without variables. Subsequently, we extended the model checker to XL (which has variables and values) by exploiting the power of the logic programming paradigm to manipulate and propagate substitutions. But the property logic has remained as the propositional (i.e., variable-free) modal mu-calculus. In this paper, we describe how the model checker in XMC can be extended to handle the value-passing modal mu-calculus, a logic that permits quantified data variables.

To date, there have been two streams of work on model checking value-passing calculus. Rathke and Hennessy [RH97] develop a local model checking

* Research supported in part by NSF grants EIA-9705998 and CCR-9876242.

algorithm, but consider constructs in the value-passing calculus that prevent any guarantees on the completeness of the algorithm. Mateescu [Mat98] also gives a local algorithm, but for the alternation-free fragment of the calculus. The performance of the algorithms are not discussed in either work, but Mateescu's algorithm has been incorporated in the CADP verification toolkit [FGK+96].

In contrast to these two works, we consider a relatively simple but still expressive set of value-passing constructs in the calculus, and describe a model checker that can be constructed with very minor changes to the propositional model checker. It should be noted that our model checker is also local. Furthermore, the extensions proposed here can be applied to mu-calculus formulas of arbitrary alternation depth, and that too with little overhead for handling the propositional fragment of the logic. This is yet another illustration of the expressiveness of the logic programming paradigm and its potential for improving the state of the art in an important application area.

The rest of the paper is organized as follows. We begin with a description of modal mu-calculus, its semantics and its model checker as a logic program (Section 2). We then introduce the value-passing modal mu-calculus and its semantics (Section 3) and describe the changes needed in the model checker to support value passing (Section 3.3). We show that the performance of the model checker for the value-passing calculus matches the performance of the original model checker for the propositional case (Section 3.4). The work on value-passing logics raises interesting issues on model checking formulas with *free* variables, which can be used to query the models (Section 4).

2 Propositional Modal Mu-calculus

The modal mu-calculus [Koz83] is an expressive temporal logic whose semantics is usually described over sets of states of *labeled transition systems* (LTSs). An LTS is a finite directed graph with nodes representing states, and edges representing transitions between states. In addition, the edges are labeled with an *action*, which is a symbol from a finite alphabet. The LTS is encoded in a logic program by a set of facts `trans(`*Src*`,` *Act*`,` *Dest*`)`, where *Src*, *Act*, and *Dest* are the source state, label and target state, respectively, of each transition.

Preliminaries: We use the convention used in logic programming, writing variable names in upper case, function and predicate names in lower case. We use θ to denote substitutions, which map variables to terms over a chosen signature (usually the Herbrand domain). By $\{X \leftarrow t\}$ we denote a substitution that maps variable X to term t. Application of a substitution θ to a term t is denoted by $t[\theta]$, and composition of substitutions is denoted by '$\circ$'.

2.1 Syntax

Formulas in the modal mu-calculus are written using the following syntax:

$$\varphi \longrightarrow Z \mid tt \mid ff \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \langle A \rangle \varphi \mid [A]\varphi \mid \mu Z.\varphi \mid \nu Z.\varphi$$

In the above, Z is drawn from a set of formula names and A is a *set* of actions; tt and ff are propositional constants; $\vee$ and $\wedge$ are standard logical connectives; and $\langle A\rangle\varphi$ (possibly after action A formula φ holds) and $[A]\varphi$ (necessarily after action A formula φ holds) are modal operators. The formulas $\mu Z.\varphi$ and $\nu Z.\varphi$ stand for least- and greatest- fixed points respectively.

For example, a basic property, the absence of deadlock, is expressed in this logic by the following formula:

$$\nu\mathsf{df}.[-\{\}]\mathsf{df} \wedge \langle -\{\}\rangle tt \tag{1}$$

where '$-$' stands for set complement (and hence '$-\{\}$' stands for the universal set of actions). The formula states that from every reachable state ($[-\{\}]\mathsf{df}$) a transition is possible ($\langle -\{\}\rangle tt$)

Fixed points may be nested. For instance, the property that a 'b' action is eventually possible from each state is written as:

$$\nu\mathsf{ib}.[-\{\}]\mathsf{ib} \wedge \langle -\{\}\rangle(\mu\mathsf{eb}.\langle\{b\}\rangle tt \vee \langle -\{b\}\rangle\mathsf{eb}) \tag{2}$$

The inner fixed point (involving the formula name eb) states that a 'b' transition is eventually reachable, and the outer fixed point (involving the formula name ib) asserts that the inner formula is true in all states.

Apart from nesting, the inner fixed point may refer to the formula name defined by the outer fixed point. Such formulas are called alternating fixed points. For instance, the property that a 'c' action is enabled infinitely often on all infinite paths is written as:

$$\nu\mathsf{ax}.\mu\mathsf{ay}.[-\{\}]((\langle\{c\}\rangle tt \wedge \mathsf{ax}) \vee \mathsf{ay}) \tag{3}$$

2.2 Semantics

Given the above syntax of mu-calculus formulas, we can talk about free and bound formula names. For instance, in the alternating fixed point formula given above, consider the inner least fixed point subformula: in that subformula, the name ax occurs free and ay is bound. To associate a meaning with each formula, we consider environments that determine the meanings of the free names in a formula.

A mu-calculus formula's semantics is given in terms of a set of LTS states. The environments map each formula name to a set of LTS states. We denote the semantics of a formula φ as $[\![\varphi]\!]_\sigma$, where σ is the environment.

The formal semantics of propositional mu-calculus is given in Figure 1. The set $\mathcal{U}$ denotes the set of all states in a given LTS, whose transition relation is represented by `trans/3`. The equations defining the semantics of boolean operations, as well as the existential and universal modalities are straightforward. The least fixed point is defined as the intersection (i.e., the smallest) of all the pre-fixed points, while the greatest fixed point is defined as the union (i.e, the largest) of all post-fixed points.

$$
\begin{aligned}
[\![tt]\!]_\sigma &= \mathcal{U} \\
[\![ff]\!]_\sigma &= \{\ \} \\
[\![Z]\!]_\sigma &= \sigma(Z) \\
[\![\varphi_1 \vee \varphi_2]\!]_\sigma &= [\![\varphi_1]\!]_\sigma \cup [\![\varphi_2]\!]_\sigma \\
[\![\varphi_1 \wedge \varphi_2]\!]_\sigma &= [\![\varphi_1]\!]_\sigma \cap [\![\varphi_2]\!]_\sigma \\
[\![\langle A \rangle \varphi]\!]_\sigma &= \{s \mid \exists a \in A \text{ such that } \texttt{trans}(s,a,t) \text{ and } t \in [\![\varphi]\!]_\sigma\} \\
[\![[A]\varphi]\!]_\sigma &= \{s \mid \forall a \in A, \texttt{trans}(s,a,t) \Rightarrow t \in [\![\varphi]\!]_\sigma\} \\
[\![\mu Z.\varphi]\!]_\sigma &= \cap\{S \mid [\![\varphi]\!]_{\{Z \leftarrow S\} \circ \sigma} \subseteq S\} \\
[\![\nu Z.\varphi]\!]_\sigma &= \cup\{S \mid S \subseteq [\![\varphi]\!]_{\{Z \leftarrow S\} \circ \sigma}\}
\end{aligned}
$$

Fig. 1. Semantics of propositional modal mu-calculus

2.3 Model checker as a logic program

We now describe how a model checker for modal mu-calculus can be encoded as a logic program. We first outline the representation of mu-calculus formulas, and then derive a model checker based on the semantics in Figure 1.

Syntax We represent modal mu-calculus formulas by a set of fixed point *equations*, analogous to the way in which lambda-calculus terms are presented using the combinator notation. We denote least fixed point equations by `+=` and greatest fixed point equations by `-=`. We also mark the *use* of formula names by enclosing it in a `form(·)` constructor. The syntax of encoding of mu-calculus formulas is given by the following grammar:

$$
\begin{aligned}
F &\longrightarrow \texttt{form(}Z\texttt{)} \mid \texttt{tt} \mid \texttt{ff} \mid F\ \texttt{\textbackslash/}\ F \mid F\ \texttt{/\textbackslash}\ F \mid \texttt{diam(}A\texttt{,}\ F\texttt{)} \mid \texttt{box(}A\texttt{,}\ F\texttt{)} \\
D &\longrightarrow Z\ \texttt{+=}\ F \quad \text{(least fixed point)} \\
&\quad \mid\ Z\ \texttt{-=}\ F \quad \text{(greatest fixed point)}
\end{aligned}
$$

Nested fixed point formulas are encoded as a set of fixed point equations. For instance, the nested fixed point formula for "always eventually b" (Formula (2)) is written in equational form as:

```
ib -= box(-{}, form(ib)) /\ diam(-{}, form(eb))          (4)
eb += diam({b}, tt) \/ diam(-{b}, form(eb))
```

Alternating fixed point formulas can be captured in equational form by explicitly parameterizing each fixed point by the enclosing formula names. For instance, the property "infinitely often c" (Formula (3)) is written in equational form as:

```
ax     -= form(ay(form(ax)))                              (5)
ay(AX) += box(-{}, (diam({c}, tt) /\ AX)) \/ form(ay(AX))
```

Semantics Based on the semantics in Figure 1, a model checker for propositional modal mu-calculus can be encoded using a predicate `models/2` which verifies whether a state in a LTS models a given formula. For encoding the semantics note that the existential quantifier and the least fixed point computation can be inherited directly from the Horn clause notation and tabled resolution (minimal model computation) respectively. The encoding of a model checker for this sublogic is given in Figure 2.

```
models(State_S, tt).

models(State_S, (F1 \/ F2)) :-
       models(State_S, F1) ;
       models(State_S, F2).

models(State_S, (F1 /\ F2)) :-
       models(State_S, F1),
       models(State_S, F2).

models(State_S, diam(As, F)) :-
       trans(State_S, Action, State_T),
       member(Action, As),
       models(State_T, F).

models(State_S, form(FName)) :-
       FName += Fexp,
       models(State_S, Fexp).
```

Fig. 2. A model checker for a fragment of propositional mu-calculus

In order to derive a model checker for the remainder of the logic, two key issues need to be addressed: (i) an encoding of the '$\forall$' quantifier which is used in the definition of the universal modality, and (ii) a mechanism for computing the greatest fixed points.

Encoding greatest fixed point computation: The greatest fixed point computation can be encoded in terms of its dual least fixed point computation using the following identity:

$$\nu Z.\varphi \equiv \neg\mu Z'.\neg\varphi[\{Z \leftarrow \neg Z'\}]$$

For alternation-free mu-calculus formulas, observe that the negations between a binding occurrence of a variable and its bound occurrence can be eliminated by reducing the formula to negation normal form. For instance, the nested fixed point formula above (Formula (4)) can be encoded using least fixed point operators and negation (denoted by `neg(·)`) as:

```
ib  += neg(form(nib))
nib += diam(-{}, form(nib)) \/ box(-{}, neg(form(eb))
eb  += diam({b}, tt) \/ diam(-{b}, form(eb))
```

Thus, there are no cycles through negation for alternation-free formulas. For formulas with alternation, the transformation introduces cycles through negation. For instance, the alternating fixed point formula originally encoded as

```
ax     -= form(ay(form(ax)))
ay(AX) += box(-{}, (diam({c}, tt) /\ AX)) \/ form(ay(AX))
```

can be encoded using least fixed point operators and negation as

```
ax     += neg(form(nax))
nax    += neg(form(ay( neg(form(nax)) )))
ay(AX) += box(-{}, (diam({c}, tt) /\ AX)) \/ form(ay(AX))
```

Note the cycle through negation in the definition of `nax` that cannot be eliminated. Given that all the currently-known algorithms for model checking alternating formulas are exponential in the depth of alternation, it appears highly unlikely that there is some formulation of this problem in terms of negation that avoids negative cycles.

Cycles and Negation: Logic programs where predicates are not (pairwise) mutually dependent on each other via negation are called *stratified*. A stratified program has a unique least model which coincides with its well-founded model [vRS91], as well as its stable model [GL88]. A non-stratified program (i.e., where there are cycles through negation) may have multiple stable models or none at all; whereas it has a unique (possibly three-valued) well-founded model. For non-stratified programs, well founded models can be computed in polynomial time [CW96], while determining the presence of stable models is NP-complete. Hence, we avoid cycles through negation in our encoding wherever possible.

```
forall(_BoundVars, Antecedent, Consequent) :-
    bagof(Consequent, Antecedent, ConsequentList),
    all_true(ConsequentList).

all_true([]).
all_true([Goal | Rest]) :-
    call(Goal),
    all_true(Rest).
```

Fig. 3. Implementing `forall/3` using Prolog builtins

Encoding the universal quantifier: The universal quantifier can be cast in terms of its dual existential quantifier using negation, but this can introduce cycles through negation even for alternation-free mu-calculus formulas. For instance, in the formula expressing deadlock-freedom property (Formula (1)) replacing the box-modality with its dual using negation will result in the following encoding:

```
df -= neg( diam(-{}, neg(form(df))) ) \/ diam(-{}, tt)
```

Hence we retain the box-modality and use an explicit programming construct `forall/3` to encode the model checker. The `forall/3` construct can itself be implemented using other Prolog builtins as shown in Figure 3. It should be noted that the above implementation of `forall/3` is correct only when there are no free variables. In the presence of free variables, one needs to keep track of their substitutions, and in general, disequality constraints on their substitutions. However, for the mu-calculus model checkers considered in this paper, we can ensure that there are no free variables in any use of `forall/3`, and hence this simple implementation suffices. When there are no free variables, we do not need to keep track of bound variables either, and hence the Prolog variable *BoundVars* in the above implementation is treated like an anonymous variable.

The encoding of the model checker for the remainder of the logic is shown in Figure 4. Taken together with Figure 2, the encoding reduces model checking to logic-program query evaluation: verifying whether a state S models a formula F is done by issuing the query `models(`S`,`F`)`. By using a goal-directed query evaluation mechanism, we ensure that the resultant model checker explores only a portion of the state space that is sufficient to prove or disprove a property [RRR+97].

3 Value-Passing Modal Mu-Calculus

We consider value-passing modal mu-calculus where the actions in modalities may be terms with data variables, and the value of these variables can be "tested" using predicates. The quantification of a data variable is determined by the modality in which it first appears. A data variable bound by a diamond modality is existentially quantified, and that bound by a box modality is universally quantified. The scope of the modality is same as the scope of the quantification.

The presence of data variables makes property specification concise, and, more importantly, independent of the underlying transition system. For instance, consider the specification of a property of a protocol that states that every message sent will be eventually received, where different messages are distinguished by identifiers ranging from 1 to some integer n. This property is expressed by the formulas in Figure 5, encoded in logics without (non-value-passing) and with (value-passing) data variables. Consider the non-value-passing case with the number of distinct messages (i.e., n in the figure) is two. The formula `str` states that after every `s1` (i.e., send of message 1) `rcv1` must hold; and after

```
models(State_S, box(As, F))  :-
       forall(State_T,
          (trans(State_S, Action, State_T), member(Action, As)),
          models(State_T, F)).

models(State_S, form(FName)) :-
       FName -= Fexp,
       negate(Fexp, NegFexp),
       tnot(models(State_S, NegFexp)).

models(State_S, neg(form(Fname))) :-
       FName += Fexp,
       tnot(models(State_S, Fexp)).

models(State_S, neg(form(Fname))) :-
       FName -= Fexp,
       negate(Fexp, NegFexp),
       models(State_S, NegFexp).

negate(tt, ff).
negate(ff, tt).
negate(F1 /\ F2, G1 \/ G2) :- negate(F1, G1), negate(F2, G2).
negate(F1 \/ F2, G1 /\ G2) :- negate(F1, G1), negate(F2, G2).
negate(diam(A, F), box(A, G)) :- negate(F, G).
negate(box(A, F), diam(A, G)) :- negate(F, G).
negate(form(FName), neg(form(FName))).
```

Fig. 4. A model checker for the remainder of the propositional mu-calculus

Without Value-passing:

```
str -= box({s1}, form(rcv1))
                  /\ box({s2}, form(rcv2)) ···
                  /\ box({s_n}, form(rcv_n))
               /\ box(-{s1, s2, ..., s_n}, form(str))
rcv1 += box(-{r1}, form(rcv1))
rcv2 += box(-{r2}, form(rcv2))
⋮
rcv_n += box(-{r_n}, form(rcv_n))
```

With Value-passing:

```
str -= box({s(X)}, form(rcv(X))) /\
                          box(-{s(_)}, form(str))
rcv(X) += box(-{r(X)}, form(rcv(X)))
```

Fig. 5. Mu-calculus formulas with and without value-passing

every `s2` (i.e., send of message 2) `rcv2` must hold; and after every non-send action `str` itself holds. The formula `str` is a greatest fixed point equation since the property holds on all (infinite) paths of evolution of the system that contain no sends. The formula `rcv1` (`rcv2`) states that the action `r1` (`r2`) is eventually enabled on all evolutions of the system.

Note that in the non-value-passing case, we have to enumerate the $\texttt{rcv}_i$ for each i. In contrast, we can state the property in a value-passing logic with a single formula using variables quantified over [1,n]. As the example shows, the formula need not even specify the domain of quantification if the underlying system generates only values in that domain.

3.1 Syntax

Before we describe the semantics of the value-passing logic, we extend our definition of LTSs, to include labels that are drawn from a finite set of *ground terms* (i.e., terms without variables) instead of just atoms. Similarly, we extend the syntax of the logic to have actions in modalities range over arbitrary (ground or nonground) terms. The syntax of the value-passing logic is thus extended as follows.

We divide the set of symbols into four disjoint sets: variables, predicate symbols, function symbols, and formula names. Terms built over these symbols are such that the formula names and predicate symbols occur only at the root. Let F represent terms with function names at root; P represent terms with predicate symbols at root; $\mathcal{A}$ represent a set of terms with function symbols at root. The set of predicate symbols include the two base propositions tt and ff. Then value-passing mu-calculus formulas are given by the following syntax:

$$\psi \longrightarrow F \mid P \;\mid\; \psi \vee \psi \;\mid\; \psi \wedge \psi \;\mid\; \langle \mathcal{A} \rangle \psi \;\mid\; [\mathcal{A}]\psi \;\mid\; \mu F.\psi \;\mid\; \nu F.\psi$$

A variable that occurs for the first time in a modality is bound at that occurrence. The scope of the binding spans the scope of the modality. In the following, we consider only value-passing formulas that are closed: i.e., those that contain no free variables.

3.2 Semantics

As in the propositional case, the semantics of value-passing formulas is given in terms of a set of LTS states. However, due to the presence of data variables, we split the environment into two parts: σ that maps each formula name to a set of LTS states, and θ that maps each data variable to a term. We denote the semantics of a formula ψ as $[\![\psi]\!]_{\sigma,\theta}$, where σ, θ are the pair of environments. The semantics of value-passing modal mu-calculus is given in Figure 6. The salient aspect of value passing in the logic is that the values of data variables are picked up from the labels in the underlying LTS. This is captured in the semantics of diamond and box modalities by picking up a substitution θ' that matches an action in the set $\mathcal{A}$ with some label in the LTS, and evaluating the remaining formula under the effect of this substitution (i.e., $\theta' \circ \theta$).

$$
\begin{aligned}
[\![P]\!]_{\sigma,\theta} &= \begin{cases} \mathcal{U} & \text{if } P\theta \text{ is true} \\ \{\} & \text{if } P\theta \text{ is false} \end{cases} \\
[\![F]\!]_{\sigma,\theta} &= \sigma(F) \\
[\![\psi_1 \vee \psi_2]\!]_{\sigma,\theta} &= [\![\psi_1]\!]_{\sigma,\theta} \cup [\![\psi_2]\!]_{\sigma,\theta} \\
[\![\psi_1 \wedge \psi_2]\!]_{\sigma,\theta} &= [\![\psi_1]\!]_{\sigma,\theta} \cap [\![\psi_2]\!]_{\sigma,\theta} \\
[\![\langle\mathcal{A}\rangle\psi]\!]_{\sigma,\theta} &= \{s \mid \exists a \in \mathcal{A}, \text{ and substitution } \theta' \text{ such that} \\
&\qquad \texttt{trans(}s\texttt{,}a\theta'\texttt{,}t\texttt{)} \text{ and } t \in [\![\psi]\!]_{\sigma,\theta'\circ\theta}\} \\
[\![[\mathcal{A}]\psi]\!]_{\sigma,\theta} &= \{s \mid \forall a \in \mathcal{A}, \text{ and substitution } \theta' \text{ such that} \\
&\qquad \texttt{trans(}s\texttt{,}a\theta'\texttt{,}t\texttt{)} \Rightarrow t \in [\![\psi]\!]_{\sigma,\theta'\circ\theta}\} \\
[\![\mu F.\psi]\!]_{\sigma,\theta} &= \cap\{S \mid [\![\psi]\!]_{\{F\leftarrow S\}\circ\sigma,\theta} \subseteq S\} \\
[\![\nu F.\psi]\!]_{\sigma,\theta} &= \cup\{S \mid S \subseteq [\![\psi]\!]_{\{F\leftarrow S\}\circ\sigma,\theta}\}
\end{aligned}
$$

Fig. 6. Semantics of value-passing modal mu-calculus

3.3 Model Checking the Value-Passing Modal Mu-Calculus

We first extend the syntax of our encoding of mu-calculus (Section 2.3) by replacing the propositions `tt` and `ff` by the more general `pred(`P`)` where P is a term representing a predicate (e.g., `pred(X=Y)`). Note that, in our encoding, formula names are already terms (to accommodate parameters used for alternating fixed points).

From the semantics of the propositional calculus (Figure 1) and that of the value-passing calculus (Figure 6), observe that a model checker for the value-passing case needs to (i) maintain and propagate the substitutions for data variables; and (ii) evaluate predicates defined over these variables. Note that when these semantic equations are implemented by a logic programming system, no additional mechanism is needed to propagate the substitutions on data variables. In other words, an environment that maps variables to values is already maintained by a logic programming engine. Hence, a model checker for the value-passing case can be derived from that for the propositional case by replacing the `tt` rule (first clause in Figure 2) with the following rule:

```
models(State_s, pred(Pred)) :- call(Pred).
```

Also, note that since the labels of an LTS are *ground* terms, the query `trans(`s`, `a`, `t`)` for any given s is always safe; i.e., a and t are ground terms upon return from the query. This ensures that every call to `models/2` is ground, provided the initial query is ground. Hence, the model checker for the value passing calculus can be evaluated using *without the need for constraint processing* on any logic programming system that is complete for programs with bounded term-size property. In fact, since every call to `models/2` is ground, there are no free variables to worry about when using the `forall` construct— we can simply use the implementation shown in Figure 3.

3.4 Experimental Results

From the relatively minor change to the definition of `models/2`, it is easy to see that the performance of the model checker for the value-passing calculus, when used on a propositional formula, will be no worse than the specialized model checker for the propositional case. The interesting question then is to compare the performance of the two model checkers on formulas that can be expressed in the propositional calculus but more compactly in the value-passing calculus (e.g., see Figure 5).

Table 1 summarizes the performance of the value-passing model checker and the propositional model checker on the property shown in Figure 5. The measurements were taken on a 600MHz Pentium III with 256MB running Linux 2.2. We checked the validity of that property for different values of domain size (n in that figure) on a specification of a two-link alternating bit protocol (ABP). The two-link version of the protocol is obtained by cascading two ABP specifications, connecting the receiver process of one link to the sender of the next. We chose the two-link version since the single-link ABP is too small for any meaningful performance measurement.

The space and time performance from the table shows that the value-passing model checker performs as well as the propositional one; the difference in speeds can be attributed to encoding used in the propositional formula, where a modality with a variable is expanded to a sequence of explicit conjunctions or disjunctions. More experiments are needed to determine whether the succinctness of value-passing formula does indeed have an impact on performance.

Domain Size	Propositional MC		Value-passing MC	
	Time	Space	Time	Space
2	4.6s	4.3M	4.3s	4.5M
3	12.9s	8.2M	11.6s	8.5M
4	24.1s	13.0M	20.9s	13.5M
5	39.8s	19.1M	32.6s	19.8M
6	56.7s	26.2M	47.5s	27.0M

Table 1. Performance of propositional and value-passing model checkers

4 Conclusions and Future Work

We showed how the power of logic programming for handling variables and substitutions can be used to implement model checkers for value-passing property logics with very little additional effort and performance penalty.

Two crucial— although common— restrictions we placed on the property logics and transition systems contributed to this simplicity. First, we considered only *closed* formulas in the property logics. This restriction is also placed in

the other works on value-passing logics [RH97,Mat98]. With this restriction, we ensured that the model checking query still produces only an yes/no answer. The interesting problem that we are currently investigating is whether model checking can truly be a *query*: i.e., find substitutions for free variables in the property formula that make the property true. This problem is inspired by the recent work on temporal logic queries [Cha00]. It turns out (as is to be expected) that the context in which a variable occurs in a formula dictates whether that query evaluation can be done without the use of additional mechanisms such as constraint handling.

Secondly, we considered only *ground* transition systems, where the states of the system and labels on the transitions were ground terms. Most verification tools allow only ground transition systems to be described, so this is not an unusual restriction. For instance, Mateescu [Mat98] considers only ground transition systems in the context of value-passing logics; Rathke and Hennessy [RH97], on the other hand, considers nonground systems. Ground transition systems meant that model checker would terminate even for value-passing logics, since the substitutions for variables in the logics were picked up from the terms occurring in the transition system. Relaxing this restriction, to allow *symbolic* transition systems where transition labels and states may be nonground terms, will allow us to model check individual modules of a specification. This can be done either by allowing the variables in the property logic to be typed (from finite types, to ensure termination), or by evaluating the model checker under a constraint logic programming system.

In this paper, we focused on aspects of value-passing that can be handled without constraint processing machinery. This work, as well as our concurrent work on bisimulation of value-passing systems [MRRV00], indicate that verification tools for general value-passing system can be built by judiciously combining tabulation and constraint processing. The performance implications of using constraints, however, remain to be explored.

References

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2), 1986.

[Cha00] William Chan. Temporal logic queries. In *Computer Aided Verification (CAV)*, 2000.

[CW96] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, January 1996.

[FGK+96] J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP (CAESAR/ALDEBERAN development package): A protocol validation and verification toolbox. In *Computer Aided Verification (CAV)*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440, 1996.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *International Conference on Logic Programming*, pages 1070–1080, 1988.

[Koz83] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[Mat98] R. Mateescu. Local model checking of an alternation-free value-based modal mu-calculus. In *Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 1998.

[Mil89] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[MRRV00] M. Mukund, C. R. Ramakrishnan, I. V. Ramakrishnan, and R. Verma. Symbolic bisimulation using tabled constraint logic programming. In *International Workshop on Tabulation in Parsing and Deduction (TAPD)*, 2000.

[RH97] J. Rathke and M. Hennessy. Local model checking for value-passing processes. In *International Symposium on Theoretical Aspects of Computer Software*, 1997.

[RRR+97] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. L. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *Proceedings of the 9th International Conference on Computer-Aided Verification (CAV '97)*, Haifa, Israel, July 1997. Springer-Verlag.

[RRS+00] C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, Y. Dong, X. Du, A. Roychoudhury, and V.N. Venkatakrishnan. XMC: A logic-programming-based verification toolset. In *Computer Aided Verification (CAV)*, 2000. XMC is available from `http://www.cs.sunysb.edu/~lmc`.

[vRS91] A. van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3), 1991.

[XSB] XSB. The XSB logic programming system. Available from `http://xsb.sourceforge.net`.