SOFTWARE VERIFICATION RESEARCH CENTRE

SCHOOL OF INFORMATION TECHNOLOGY

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 01-16

Model Checking with Abstract Types

Kirsten Winter

Version 1, November 2001

Phone: +61 7 3365 1003
Fax: +61 7 3365 1533
http://svrc.it.uq.edu.au

To appear in *Electronic Notes in Theoretical Computer Science, vol. 55 No. 3 (2001).*

Model Checking with Abstract Types

Kirsten Winter

SVRC, University of Queensland, Queensland 4072 Australia
email: *kirsten@svrc.uq.edu.au* phone: +61 7 3365 1656

Abstract

Model checking the design of a software system can be supported by providing an interface from a high-level modelling language, which is suitable for describing software design, to a given model checking tool. In order to cope with the higher complexity of software systems, we additionally need a means for reducing the system's state space. This can be done be applying abstraction to large or infinite data parts of the model. In our work, we introduce an interface from the high-level modelling language ASM to Multiway Decision Graphs (MDGs). Similar to OBDDs, MDGs are a data structure suitable for symbolic representation of transition systems, and their model checking. Since MDGs support the representation of abstract sorts and functions they can treat abstract models. We present a transformation algorithm from ASM to MDGs that automatically generates abstract models once the user has marked the data to be abstracted. We adapt the MDG model checking algorithms for the treatment of ASM models.

1 Introduction

Most model checkers operate on models that are given in a low-level language that is developed for specifying hardware circuits rather than software. The development of software, however, should be supported by a high-level language that provides usual language facilities such as complex data types, and appropriate structuring mechanisms. When using model checking to support analysing software models, we have to provide a transformation from such a high-level language to the input language of a chosen model checker. Such a transformation has to bridge the gap between the different languages in a semantic preserving way that, moreover, avoids adding complexity to the model checking task as far as possible.

In general, software systems involve more complexity than hardware systems due to their more complex data part. One means to cope with this complexity is *abstraction*: We may reduce the state space of a model by applying an abstraction function that provides a lifting of infinite sets into finite sets of equivalence classes that are relevant for the behaviour of the model. Two additional tasks arise within this approach: An appropriate abstraction function has to be defined and the abstract model that preserves the properties to be checked has to be computed.

In our work, we use *Multiway Decision Graphs* (MDGs) (see [CZS$^+$97, CCL$^+$97]) as a basic data structure for implicit state enumeration and model checking ([XCS$^+$98]). Similar to the OBDD approach that is used for SMV ([McM93]) and other tools, MDGs are suitable for representing state transition systems and provide efficient algorithms for computing and checking their reachable states. In contrast to OBDDs, however, MDGs are able to represent values of any enumeration set (rather than being restricted to boolean values) and support the use of *abstract* sorts and functions. This provides a means of avoiding expensive boolean encoding (as in OBDD-based approaches) and to treat possibly infinite systems.

In [XCS$^+$98], MDGs are used for model checking circuit designs given in a hardware description language, called MDG-HDL. In our approach, we aim at using the MDG approach for supporting a more general specification language, called *Abstract State Machines* (ASM, [Gur95]). ASM has been used in academic and industry contexts to develop systems in a variety of domains (a bibliography is given in [Hug]). ASM model transition systems in a simple and uniform fashion and provide an operational semantics. A transformation from ASM into the input language of a transition system-based model checker appears feasible as first results with the SMV tool show (see [Win97, CW00]).

By representing ASM models by means of MDGs, we benefit from a straightforward transformation (similar to the approach used for interfacing the SMV tool) and gain a very simple means for generating abstract models: We extend the ASM language with a notion of abstract types. If the user chooses an infinite or large data type to be abstract then consequently all functions that are applied to this type turn into abstract functions that are left uninterpreted during the state exploration. Predicates over abstract data automatically provide a suitable partitioning into equivalence classes. This different treatment of abstract functions and predicates is provided automatically during our transformation step from ASM into MDGs. An implementation of the transformation algorithm is available.

Due to the corresponding paradigm of both notations and the aim of efficient representation, we do not map ASM onto the given hardware description language MDG-HDL but rather provide a mapping from ASM to the MDG data structure itself. Consequently, we are not supported with a black box model checking tool that is ready to use but rather with a library of all necessary functions for computing MDGs. That is, we rebuild a model checker for the needs of ASM based on this library according to the work that is done for model checking MDG-HDL models ([Xu99, XCS$^+$98]). Apart from the mapping of ASM to MDGs, we therefore have to adapt some algorithms that are developed for model checking circuits.

An outline of the paper follows: Section 2 and Section 3 introduce ASM and MDGs, respectively. In Section 4, we show our transformation from ASM into the graph structure of MDGs. Our model checking approach for ASM using MDGs is introduced in Section 5. We conclude this work with related and future work in Sections 6 and 7.

2 Abstract State Machines

Abstract State Machines (ASM) is a high-level language suitable for a wide range of domains (for a bibliography see [Hug]). It is a state-based approach for describing transition systems: The states are given in terms of sorts (domains) and functions over these sorts. Functions can be *static* and have a constant value, *dynamic* and thus be controlled by the system during a run (similar to state variables), or *external* and controlled by the environment (like input variables). The behaviour of the system is given as a set of transition rules which specifies the *updating* of the dynamic functions depending on the current state of the system. The ASM notation provides a `skip` rule, a simple `update` rule that changes the value of (i.e., updates) a dynamic function, a `block` rule that gathers a set of rules, a `conditional` rule that restricts a rule to fire only in states that satisfy the guarding condition (which is a first-order predicate), a `do-forall` rule that applies a parameterised rule for all members of a given set, and a `choose` rule that introduces non-determinism by applying a parameterised rule for one arbitrarily chosen element of a given set[1]. During a run of the system, all transition rules fire simultaneously, i.e., all rules are applied in a single step and lead to the next state of the system. For a full definition of ASM see ,e.g., [Gur95].

We give a short impression of the language by means of a simple example, a generic timer. Figure 1 sketches the behaviour of the system: It gets a natural number *max* as an input that specifies the number the timer has to count to. The timer has two states `COUNT`, which is the initial state, and `RING`. As long as it is in state `COUNT` it increments the value of an internal variable t. If t reaches *max*, the system changes to state `RING`, indicating that the time is elapsed. It then resets t and changes back into the initial state `COUNT`.

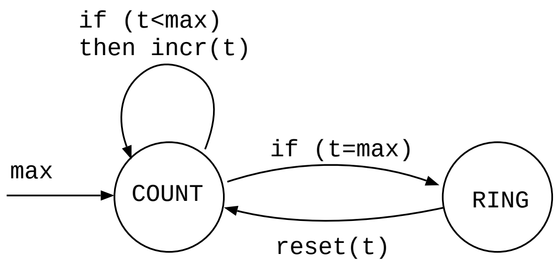

Figure 1: Example of a generic timer

When modelling the generic timer in ASM, we introduce two sorts, $\texttt{MODE} = \{count, ring\}$ and $\texttt{TIME} = \mathcal{N}$, and two dynamic functions over these, namely `mode:MODE` and `t:TIME`. Initially, `mode` equals *count* and `t` equals 0. To model the input, we introduce an external function `max:TIME`. Two static functions model incrementing and reseting of the timer variable, $\texttt{incr} = \{t \mapsto (t+1) \mid t \in \texttt{TIME}\}$ and $\texttt{reset} = \{t \mapsto 0 \mid t \in \texttt{TIME}\}$. The ASM model is now given as in Figure 2.

[1]ASM supports also an import rule that allows the user to introduce fresh elements. However, this extension of sorts cannot be treated by our model checking approach.

```
typealias TIME == $INT$
freetype MODE  == $\{count, ring\}$
static function    $n$      == 100
static function    Time     == $\{1..n\}$
external function  **max**    :  TIME
static function    **reset**  :  TIME $\rightarrow$ TIME  ==  $\{t \mapsto 0 \mid t \in$ Time$\}$
static function    **incr**   :  TIME $\rightarrow$ TIME  ==  $\{t \mapsto (t+1) \mid t \in$ Time$\}$
dynamic function   **mode**   :  MODE  initially  *count*
dynamic function   **timer**  :  TIME  initially  0

transition R1  ==   if (mode = count) $\wedge$ (timer < max)
                      then  timer := incr(timer)

transition R2  ==   if (mode = count) $\wedge$ (timer = max)
                      then  mode := ring

transition R3  ==   if (mode = ring)
                      then  mode := count
                            timer := reset(timer)
```

Figure 2: The concrete ASM model for the timer

A run of this ASM model is defined as a sequence of states starting with the initial state that is defined through the initial values of dynamic functions. In every state, the next state is derived by applying the transition rules simultaneously to the current state. (Note that updates within one rule are also fired simultaneously, see rule R3 for example.)

Abstract ASM. In order to exploit the support of abstract data types provided by MDGs, we introduce a syntactic feature to label any sort as being an *abstract sort*. In the example given above, we might change the sort TIME to be abstract.

Functions over abstract sorts do not have a fixed interpretation. They allow for any interpretation that matches their signature. Abstracting from sorts is a means of lifting a "concrete" ASM model into an "abstract" ASM model whose instances comprise concrete models for all possible interpretations of the abstract sorts and functions.

Figure 3 depicts the abstraction step: We consider the sort Q in our concrete model as abstract and change all its occurrences into $Data_{abs}$. As a result, we get an abstract model of the same signature. For this abstract specification, all those interpretations are suitable that have a sort, a 2-ary function that maps arguments of the given sort to a value of the same sort, and a boolean predicate over the sort. In the figure, we give some examples for different interpretations for the sort $Data_{abs}$ and the functions that are possible.

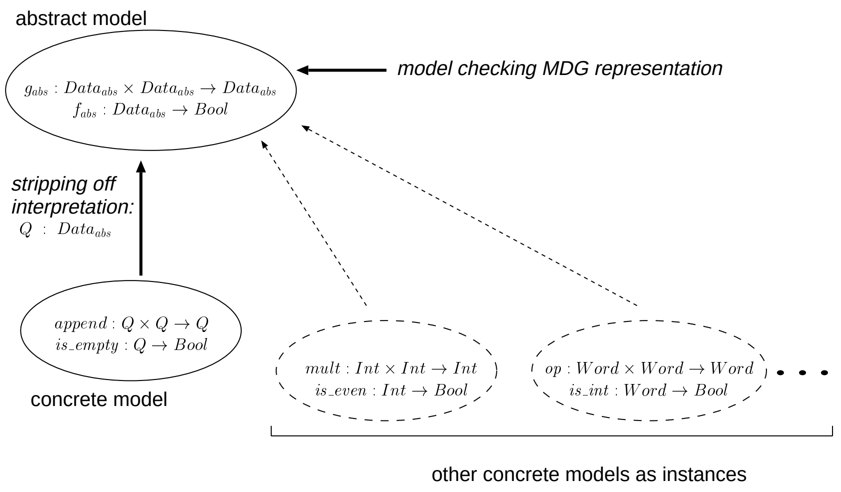
abstract model
$g_{abs} : Data_{abs} \times Data_{abs} \to Data_{abs}$
$f_{abs} : Data_{abs} \to Bool$
model checking MDG representation
stripping off interpretation:
$Q \; : \; Data_{abs}$
$append : Q \times Q \to Q$
$is\_empty : Q \to Bool$
concrete model
$mult : Int \times Int \to Int$
$is\_even : Int \to Bool$
$op : Word \times Word \to Word$
$is\_int : Word \to Bool$
other concrete models as instances

Figure 3: Lifting a concrete model to an abstract model

The purpose of this abstraction step is to substitute infinite sorts, and functions over them, since these cannot be exhaustively explored. When checking an abstract model, abstract functions (i.e., functions that map into an abstract sort) are left uninterpreted. Functions over some abstract sorts that map to a concrete finite sort (as, for instance, boolean predicates over abstract parameters) can be investigated by means of a complete case distinction. This naturally provides a partitioning of the infinite sort into finitely many equivalence classes. The state space of the abstract model is thus smaller in most cases. It can be canonically represented by MDGs as we show in the following sections.

3 Multiway Decision Graphs

Multiway Decision Graphs (MDGs) are a generalisation of Binary Decision Diagrams. They are a data structure for canonically representing formulas of a many-sorted first-order logic, called *Directed Formulas* (DFs). A special feature of the underlying logic is the distinction between *concrete* and *abstract* sorts. Correspondingly, function symbols may be concrete, abstract (if the range is abstract), or cross-operators (if the range is concrete but the domain contains some abstract sort).

DFs are suitable to describe sets of states and transition relations of transition systems. They are formulas in disjunctive normal form (DNF) over simple equations of the following form: $f(B_1, \dots B_n) = a$ (where f is a cross-operator and a is a constant of concrete sort), $w = a$ (where w is a variable of concrete sort and a is a concrete constant), or $v = A$ (where v is a variable of abstract sort and A is a term of the same sort). Furthermore, in each disjunct of a DF, all left hand sides (LHSs) of the equations are pairwise distinct and every abstract variable that occurs as a LHS must occur in every disjunct of the DF. To be represented by MDGs, a DF has to be *concretely reduced*, i.e., all concrete variables that occur on the right-hand side (RHS) of an equation have to be substituted by a value, i.e., a concrete constant.

An MDG is a finite graph G, whose non-terminal nodes are labelled by terms and whose edges, starting from a non-terminal node, are labelled by terms of the same sort as the node. Terminal nodes are labelled by formulas. Generally, a graph G represents a formula in the following way:

- If G consists of a single terminal node, then it represents the formula the node is labelled with.
- If G has a root node labelled with term A and edges labelled with terms $B_1, \ldots, B_n$ leading to subgraphs $G_1, \ldots, G_n$ that represent formulas $P_1, \ldots, P_n$, then G represents the formula

$$(A = B_1 \wedge P_1) \vee (A = B_2 \wedge P_2) \vee \ldots \vee (A = B_n \wedge P_n)$$

In Figure 4, we depict the formula that is given above as a graph G. To be a canonical representation, an MDG has to satisfy certain *well-formedness conditions* which involve an order on function symbols and variables that has to be provided by the user (the detailed list of conditions is defined in [CZS+97, CCL+97]).

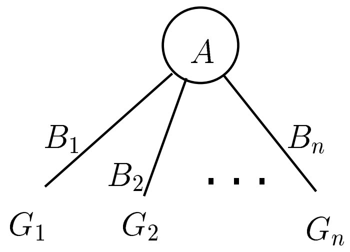

Figure 4: The MDG G

A library of operations on MDGs is available that is sufficient for realising an *implicit* state enumeration, namely disjunction, relational product, and pruning-by-subsumption. They are defined for combining sets rather than only pairs of MDGs. This allows us to represent the transition relation as a set of several small graphs instead of one big graph and benefits from a more efficient state enumeration.

The relational product operation computes the conjunction of a set of graphs under existential quantification of all variables in a given variable set E and the possible renaming η of variables, $((\exists v \in E)(\bigwedge_{1 \leq i \leq n} P_i) \cdot \eta)$. This is used for computing the set of reachable states from a given state (in which case the MDGs P_i represent the transition relation and E the set of state variables). Pruning-by-subsumption approximates the difference of two sets. This enables us to check if an invariant (given as an MDG) is satisfied in a given set of states (also given as an MDG).

According to the well-formedness conditions on MDGs, the application of the operations is restricted. The relational product of two MDGs can only be computed if their nodes are not labelled with the same abstract variable. Disjunction, in contrast, is applicable only to MDGs that contain the same set of abstract variables as nodes labels (i.e., the same set of abstract variables that occur as LHSs in the equations of the corresponding DF).

4 The Transformation from ASM to MDGs

The transformation between the two notations is split into two steps. By introducing an intermediate language, called ASM-IL, we benefit from a general interface that can be reused for connecting other model checkers or state-transition-based tools. Due to the limitation of space, we give only a short overview of the transformation algorithm. A more detailed description can be found in [Win01].

ASM into ASM-IL. Due to the fact that any set of ASM transition rules can be flattened into a set of simple guarded update rules, we can fully represent ASM models – except models that contain import-rules[2] – as a set of *location-update pairs.* A *location* is a dynamic function applied to a value, which can change its value over a run (similar to a state variable). A location-update pair comprises a location and the set of its possible guarded updates, $(loc, \{(guard_j, val_j) \mid 0 \leq j \leq n\})$. We gather this information for each location loc_i of the ASM-model. The set of all location-update pairs represents the model in ASM-IL. For example, the model of the timer as shown in Figure 2 is represented in ASM-IL in the following way:

$$\begin{aligned}\{(\textbf{timer'}, [\big(((\text{mode} = \textit{count}) \wedge (\text{timer} < \text{max})), \text{incr}(\text{timer})\big),\\ \big((\text{mode} = \textit{ring}), \text{reset}(\text{timer})\big)\ ])\\ (\textbf{mode'}, [\big(((\text{mode} = \textit{count}) \wedge (\text{timer} = \text{max})), \textit{ring}\big),\\ \big((\text{mode} = \textit{ring}), \textit{count}\big)\ ])\}\end{aligned}$$

Transforming ASM into ASM-IL involves two steps: (a) *unfolding* dynamic functions into locations if they occur as LHSs of updates (e.g., $f(x)$ with $x \in \{a, b\}$ results in f_a and f_b) or substituting them by their possible values otherwise; (b) *flattening* all (nested) transition rules into simple guarded update rules over only one location. This way, we produce a lot of code. However, predicates can be resolved and simplified due to the partial evaluation in the first step.

If this procedure meets an *abstract function* unfolding is avoided, the function is left *uninterpreted.* If it meets a function of concrete sort that is applied to abstract terms it treats this term as a *cross-term* (i.e., a cross-operator applied to some abstract terms), and avoids unfolding as well. If the cross-operator matches one of the standard relational operators (e.g., $=$, $\leq$, etc.) the tool automatically introduces a new symbol that can be distinguished (e.g., *isEq*, *leq*, etc.). For instance, the predicate $(a \leq b)$, where a and/or b are abstract terms, is mapped into $leq(a, b)$. These new cross-operators automatically introduce a partitioning of the abstract sort suitable for the model.

ASM-IL as MDGs. It can be shown that each location-update pair in ASM-IL can be represented as a well-formed MDG: Due to the fact that during the first step of our transformation we unfold every concrete function that is not on

[2]Choose rules are simulated by introducing an external variable for making the non-deterministic choice.

the LHS of an update, location-update pairs are "concretely reduced" (in MDG-terminology). Moreover, each pair contains at most one abstract variable that labels a node in the graph (these are the variables on the LHS of an equation or update), namely *loc*, the location to be updated. Any other equation in a guard that has an abstract variable as its LHS is transformed into a cross-term. We can deduce that every case in the location-update pair contains the same LHS abstract variable, namely *loc* or none.

According to the definition of MDGs, each location-update pair $(loc_i, \{(guard_{ij}, val_{ij}) \mid 0 \leq j \leq n\})$ can be represented in the graph structure as follows (where loc'_i denotes the value of loc_i in the next state): The location loc'_i labels the root node of the graph. Each edge starting at the root is labelled with one of the specified update values val_{ij} and leads to the subgraph G_{ij} that represents the corresponding guard of the update $guard_{ij}$. Figure 5 sketches a graph for a location-update pair containing three possible guarded updates. Since in ASM the state remains unchanged if none of the guards is satisfied we have to add an explicit else-case to the MDG (otherwise the next value is arbitrary). The subgraph G_{else} represents the formula $\neg(guard_{i1} \vee \ldots \vee guard_{in})$. In case loc_i is of concrete type, we have to substitute the edge-label with a path for each possible value of loc_i.

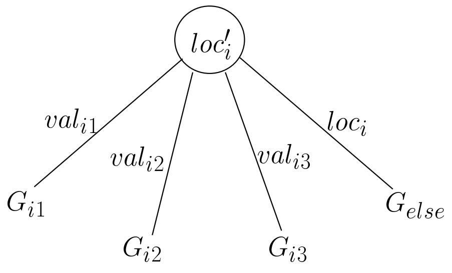

Figure 5: Location-update pair as MDG

The transformation of guards is based on the fact that simple equations $eq(lhs, rhs)$ assemble a new MDG where the root is labelled by *lhs* and the only edge, labelled with *rhs*, leads to the leaf *true*. Conjunction and disjunction can be computed by the corresponding operations on MDGs once the operands are represented as MDGs. Negation is only possible if the operand does not contain abstract node labels.

5 Adapted Model Checking Algorithm

[XCS+98, Xu99] introduce a first-order temporal logic, called $\mathcal{L}_{MDG}$, and corresponding model checking algorithms based on MDGs. $\mathcal{L}_{MDG}$ is the universal fragment (i.e., it excludes the existential temporal operator) of *Abstract_CTL**, a deviation of CTL* that has an additional construct for quantification of variables.

With the usual meaning of temporal operators, a formula in $\mathcal{L}_{MDG}$ may have the form $\mathbf{A}\varphi$, $\mathbf{AG}\varphi$, $\mathbf{AF}\varphi$, $\mathbf{A}\varphi\mathbf{U}\psi$, $\mathbf{AG}(\varphi \Rightarrow (\mathbf{F}\psi))$, or $\mathbf{AG}(\varphi \Rightarrow (\psi_1\mathbf{U}\psi_2))$, where φ, ψ, ψ_1, and ψ_2 are so called *Next_let_formulas*. The Next_let_formulas are defined as follows (where *model variables* are the input-, state- and output-variables of the system model we want to check):

- The boolean truth values `T`, `F`, and any equation $t_1 = t_2$ are

Next_let_formulas (where t_1 is a model variable and t_2 is a model variable, an ordinary variable, a function over those, or a constant).

- If p and q are Next_let_formulas then so are: $\neg p$, $p \wedge q$, $p \vee q$, $p \rightarrow q$, $\mathbf{X}p$ and **LET** $(v = t)$ **IN** p (where t is a model variable and v is an ordinary variable).

$\mathcal{L}_{MDG}$ is restricted to simple templates of formulas with temporal path operators (**A**, **AG**, **AF**, etc.) as given above and does not support nesting of those. Furthermore, the nesting of **X** in Next_let_formulas is limited. These two restrictions enable simple checking algorithms for this language ([Xu99, XCS$^+$98]): The Next_let_formulas occurring in a formula to be checked are represented as a circuit consisting of and-gates, or-gates, comparators and registers. Its output is a flag that indicates the truth-value of the Next_let_formula. This circuit is *composed* with the system model M such that M outputs those model-variables to the circuit the Next_let_formula refers to. That is, the circuit determines the truth value depending on some particular state variables of the system. The overall model checking algorithms (each formula template is treated by a particular algorithm) compute the reachable states of the composed machine and check the truth-value of the flag (if it equals true the property is satisfied).

For model checking ASM models represented as MDGs, we can use the same model checking algorithms since they are based on the MDG library. However, we have to change the representation of Next_let_formulas. They must be transformed into simple ASM models consisting of some additional variables (and their initialisation) and simple guarded transitions rules which can be transformed into ASM-IL.

Any Next_let_formula p can be represented as an ASM M_p. The composed model of system model M and M_p is then given as M extended by the additional variables and the transition rules of M_p. We build M_p in the following way: We introduce a new dynamic function $\mathit{Flag}_p : \mathit{Bool}$ which represents the truth value of p and

- if p is of the form `T`, `F` or $(t_1 = t_2)$ then it is represented by an ASM M_p with the following transition rule and initialisation of the flag variable:

  `if` p `then` $\mathit{Flag}_p := \mathit{true}$ `else` false
  initially $\mathit{Flag}_p = p$;

- if o and q are Next_let_formulas represented by Flag_o and Flag_q then the Next_let_formula p with

  - $p = \neg q$ is represented by

    `if` $\neg \mathit{Flag}_q$ `then` $\mathit{Flag}_p := \mathit{true}$ `else` false
    initially $\mathit{Flag}_p = \neg \mathit{Flag}_q$;

  - $p = o \wedge q$ is represented by

    `if` $(\mathit{Flag}_o \wedge \mathit{Flag}_q)$ `then` $\mathit{Flag}_p := \mathit{true}$ `else` false
    initially $\mathit{Flag}_p = (\mathit{Flag}_o \wedge \mathit{Flag}_q)$;

  - $p = o \vee q$ is represented by

    `if` $(\mathit{Flag}_o \vee \mathit{Flag}_q)$ `then` $\mathit{Flag}_p := \mathit{true}$ `else` false
    initially $\mathit{Flag}_p = (\mathit{Flag}_o \vee \mathit{Flag}_q)$;

– $p = o \rightarrow q$ is simplified to $p = \neg o \vee q$.

– $p = \mathbf{X}q$ is represented by

```
    if $Flag_q$ then $Flag_p := true$ else $false$
    initially $Flag_p = true$;
```

– $p = \mathbf{LET}\ (v = t)\ \mathbf{IN}\ q$ is represented by the following (where we introduce a new variable v of the same sort as t)

```
    if $((v = t) \wedge Flag_q)$ then $Flag_p := true$ else $false$
    initially $Flag_p = ((v = t) \wedge Flag_q)$;
```

In each of the given templates for M_p, the location $Flag_p$ represents the truth value of formula p. Note that for any formula $p = \mathbf{X}q$, the initialisation of $Flag_p$ equals *true*, i.e., $Flag_p$ equals the truth value of p in the next state only. For model checking $\mathbf{AG}p$, $\mathbf{AF}p$, etc. we apply the composed machine of M and M_p to the corresponding checking algorithm for $\mathbf{AG}$, $\mathbf{AF}$, etc. These algorithms basically compute the reachable states and check if $Flag_p = true$ holds always or eventually, etc. depending on the algorithm (for more detail see [Xu99, XCS$^+$98]).

6 Related Work

Uninterpreted functions are addressed elsewhere: In [BD94] data values and operations within the specification of the DLX architecture are modelled by means of uninterpreted functions. However, this approach allows only validity checking, no temporal properties can be checked. [CN94] introduce a new logic, called GTL, which also allows uninterpreted functions to be represented. The decidable fragment of GTL can be treated by an automatic validity checker (based on PVS). The logic $\mathcal{L}_{MDG}$, however, is more expressive than the decidable fragment of GTL (as proven in [Xu99]) and model checking algorithms go beyond validity checking.

Due to the support for abstract sorts in MDGs, the computation of the abstract model appears to be much simpler than the mechanisms suggested in, e.g., [GS97] and [BPR00]. Instead of providing an abstraction function and proving that properties are preserved, we generate with less effort an abstract model that includes the intended model and more. This may result in *false negatives*, that is counter-examples that are not related to the particular instance of the model we want to check (in this case, it may be possible to add rewriting rules to exclude the non-intended interpretations), but if no counter-example can be found then *all* instances are correct.

Some process algebras such as μCRL ([BP95]), or XMC ([RRS$^+$00]) allow the user to specify complex data types abstractly. These languages are also supported by model checking. They are well suited to model communicating distributed system. However, modelling a state-based software system by means of a process algebra can be quite difficult.

7 Conclusion

In this work, we presented a transformation from the high-level modelling language ASM ([Gur95]) into the data structure MDG ([CZS+97, CCL+97]). This transformation enables us to represent ASM models as MDGs and exploit their library of functions which are suitable for implementing model checking algorithms. We showed how the suggested approach for model checking $\mathcal{L}_{MDG}$([Xu99, XCS+98]), a sub-language for first-order CTL*, can be adapted to the ASM language. The benefit of this approach is a more efficient coding of ASM: Instead of forcing a binary encoding of complex data types (as within OBDD-based tools), MDGs support a more compact representation of multi-valued data types (less state variables have to be introduced). Moreover, we gain simple support for generating abstract models by means of abstracting large or infinite data types. MDGs allow for representation of abstract values and functions, of which the latter are left uninterpreted.

This approach appears to be promising for analysing software systems since complex data parts can be easily abstracted. Relational operators provide a naturally given partitioning of data types according to their influence in the control flow (e.g., whenever data are requested in the guards of a transition rule). The abstract model and the partitioning of abstract types is provided automatically by the transformation step.

The implementation of the transformation is completed. The next step is to code the model checking algorithm and show the feasibility of the approach.

References

[BD94] J. Burch and D. Dill. Automatic verification of pipelined microprocessor control. In D.L. Dill, editor, *Proc. of Int. Conf. on Computer Aided Verification, CAV'94*, volume 818 of *LNCS*, pages 68–80. Springer-Verlag, 1994.

[BP95] D. Bosscher and A. Ponse. Translating a process algebra with symbolic data values to linear format. In U.H. Engberg, K.G. Larsen, and A. Skou, editors, *Procs. of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'95*, BRICS Notes Series, pages 119–130, Aarhus, 1995.

[BPR00] T. Ball, A. Podelski, and S.K. Rajamani. Boolean and cartesian abstraction for model checking C programs. Technical Report MSR-TR-2000-115, Microsoft Research, 2000. To appear in TACAS'2001.

[CCL+97] E. Cerny, F. Corella, M. Langevin, X. Song, S. Tahar, and Z. Zhou. Automated verification with abstract state machines using Multiway Decision Graphs. In T. Kropf, editor, *Formal Hardware Verification: Methods and Systems in Comparison*, volume 1287 of *LNCS*, pages 79–113. Springer Verlag, 1997.

[CN94] D Cyrluk and P. Narendran. Ground temporal logic: A logic for hardware verification. In D. Dill, editor, *Proc. of Int. Conf. on Computer Aided Verification, CAV'94*, volume 818 of *LNCS*, pages 247–259. Springer-Verlag, 1994.

[CW00] G. Del Castillo and K. Winter. Model checking support for the ASM high-level language. In S. Graf and M. Schwartzbach, editors, *Proc. of 6th Int. Conference for Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2000*, volume 1785 of *LNCS*, pages 331–346. Springer-Verlag, 2000.

[CZS$^+$97] F. Corella, Z. Zhou, X. Song, M. Langevin, and E. Cerny. Multiway Decision Graphs for automated hardware verification. *Formal Methods in System Design*, 10(1), 1997.

[GS97] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. of Int. Conf. on Computer Aided Verification, CAV'97*, volume 1254 of *LNCS*, pages 72–83. Springer-Verlag, 1997.

[Gur95] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.

[Hug] J.K. Huggins. Abstract state machines home page. EECS Department, University of Michigan. http://www.eecs.umich.edu/gasm/.

[McM93] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[RRS$^+$00] C.R. Ramakrishnan, I.V. Ramakrishnan, S. Smolka, Y. Dong, X. Du, A. Roychoudhury, and V.N. Venkatakrishnan. XMC: A locic-programming-based verification toolset. In E.A. Emerson and A.P. Sistla, editors, *Proc. of Int. Conf. on Computer Aided Verification, CAV 2000*, volume 1855 of *LNCS*, pages 576–579. Springer-Verlag, 2000.

[Win97] K. Winter. Model Checking for Abstract State Machines. *J.UCS Journal for Universal Computer Science (special issue)*, 3(5):689–702, 1997.

[Win01] K. Winter. *Model Checking Abstract State Machines*. PhD thesis, Technical University of Berlin, Germany, http://edocs.tu-berlin.de/diss/2001/winter_kirsten.htm, 2001.

[XCS$^+$98] Y. Xu, E. Cerny, X. Song, F. Corella, and O. Ait Mohamed. Model checking for a first-order temporal logic using Multiway Decision Graphs. In *Proc. of Int. Conf. on Computer Aided Verification (CAV'98)*, volume 1427 of *LNCS*, pages 219–231. Springer-Verlag, 1998.

[Xu99] Y. Xu. *Model Checking for a First-order Temporal Logic Using Multiway Decision Graphs*. PhD thesis, University of Montreal, 1999.