Model Checking for UNITY

The UV System
Revision 1.10

Markus Kaltenbach
Department of Computer Sciences
The University of Texas at Austin

May 9, 1994

Abstract

We present a description of our current implementation of a model checker for finite state UNITY programs and propositional UNITY logic. The model checker is capable of dealing with all unconditional properties of UNITY logic. Checking safety properties and basic progress properties can be done very efficiently due to the partitioning of the transition relation of a program induced by the program statements. Finding suitable invariants remains a crucial task in proving properties. The model checker provides means for both computing the strongest invariant of a program and for managing established invariants.

Contents

1 Introduction

The UNITY Verifier (UV) project is aimed at providing automated support for the verification and the design of distributed programs. As a basic building block for such a support system we have developed the UNITY model checker (UMC), which is a tool for both automatically verifying properties of finite state UNITY programs and for supporting the design and analysis of such programs by allowing the interactive exploration and manipulation of programs and their specifications.

In working with UMC the user provides the system with a set of programs (the models) and a set of specifications. The system then checks whether the specifications are satisfied by the programs. The input language in which programs and specifications are stated allows the user to write finite state UNITY programs and propositional UNITY properties as specifications. This includes a strong typing scheme with boolean, finite range integer and enumeration types, as well as the complete treatment of all unconditional modalities of UNITY logic to express safety and progress properties.

Besides being able to completely automatically check the validity of properties with respect to given programs, the UMC system provides the user with additional information (such as counterexamples) that greatly help in debugging programs and understanding certain verification failures. Moreover the UMC system has a carefully designed user interface that supports the user in managing and investigating programs and properties. The goal for designing such an interface is to both make the system accessible to a wider range of users and to invite users to use the system to explore and to modify their programs in a way that would be to difficult or error prone if done without mechanical help.

The UMC system uses Ordered Binary Decision Diagrams (OBDDs) for internally representing formulae, sets of states, and transition relations. Programs and properties given to the system as input are compiled into OBDDs, thus resulting is a symbolic representation in which the state spaces of programs

are not enumerated or constructed explicitly. The transformation of programs, properties, and types in particular into OBDDs is transparent to the user.

It has to be noted that the UMC system is a prototype and is as such currently still under development. As a consequence of this development process there are several limitations in effect on the current system revision, that we expect to remove in future revisions. Such limitations include the lack of support for checkpointing or saving the system state beyond the level of textual descriptions in the input language, the limited implementation of tools for inspecting formulae or program execution traces, or the lack of suitable parameterization of the system in order to adjust its performance to different problems.

As the system furthermore is part of our ongoing research, there are many interesting extensions still to be investigated. These include the effective composition of several programs in a structured way (whil taking advantage of syntactic restrictions to reduce to size of the resulting state spaces), the introduction of parameterized unbounded programs and properties and the ability to prove certain formulae about such programs with the help of inductive methods, or the exploitation of symmetry to reduce the size of state spaces. It should be already clear that in the larger context of an automated support system for the design of distributed programs a model checker in the given form is a very useful but still rather basic tool that needs to be supplemented by more powerful system components in order to make such a system useful for a larger class of design tasks.

The reminder of this paper is structured as follows: in section 1 we first recall some basic concepts of model checking and UNITY, define basic structures of the UMC system, and show how they are related to each other. In section 2 we give a few simple examples of inputs to the system, and of how the system can be used on those inputs. We then proceed with a complete description of the input language in section 3. We conclude the main part of the paper by discussing some practical aspects of the UMC system and by stating our objectives for the next steps in the its future development in section 4. Two appendices contain a summary of the grammar of the input language, and a some information about the actual implementation of the UMC system.

2 Concepts and Structures

The model checking task consists in determining whether a given formula in some logic is satisfied by a structure over which the formula is interpreted. In the UMC system the formulae considered are the properties of propositional UNITY logic and the programs are finite state UNITY programs. We assume that the reader is familiar with UNITY, in particular with the programming notation and the semantics of the properties (for reference see [CM 88], [Mis 93]).

The purpose of this section is to summarize the concepts of model checking for UNITY, to explain how these concepts are dealt with in the UMC system,

and how the UMC system can be used for supporting UNITY model checking.

2.1 Documents

The user interacts with the UMC system on the basis of text documents, that contain textual representations of collections of programs and properties to be investigated. Such a document is parsed and compiled into an internal representation suitable for invoking the model checking algorithms and for performing related computations.

After parsing a document the user can access all programs and all properties of the document through an easy to use interface, can selectively display status information about each program or property, can selectively check properties with respect to different invariants, and can obtain information about these model checker invocations, that often allow to better understand, why a certain property fails to be proved valid.

2.2 Programs

UNITY programs define the structures for which properties are verified. The UMC system is capable of handling finite state UNITY programs without quantified assignments. The finiteness of the state space of an admissible program is guaranteed by the restriction on available data types, which are boolean, finite range integer, and enumeration types.

Theoretically every finite state program can be represented solely with boolean variables, but it is very cumbersome and an additional source of errors to ask the user to provide a binary encoding herself. Moreover by supporting non-boolean finite types the readability of programs is significantly increased.

The UMC input language derived from the UNITY programming notation allows a powerful and convenient way of defining a state transition system for which properties are to be proved. In a declaration part of the program the state space of the program is defined as the cartesian product of the domains of all declared variables. A second part of the program allows to specify initial states of the system, whereas the assignment part of a program consists of a collection of guarded multiple assignments defining the transition relation of the program. The notation allows to easily expresses both synchronous and asynchronous behavior, since all assignments in any multiple assignment are performed simultaneously, and since an execution of a program consists of any weakly fair sequence of assignment executions.

The UMC system associates with every program an invariant, namely the strongest invariant proved to hold for the program. This notion is well defined, since with any two invariants I and J their conjunction $I \wedge J$ is an invariant as well. Furthermore there is a strongest invariant of any program characterizing the set of states reachable from initial states by a program execution. The

significance of these invariants for the model checking task is discussed later in this section.

2.3 Properties and Model Checking

The UNITY logic is built from eight kinds of (unconditional) modalities called properties that allow the concise specification of many interesting aspects of programs. UNITY logic is a simple positive logic in the sense that logical operations (in particular negation) or nesting of modalities are not part of the logic. The modalities are applied to nonmodal formulae, in the UMC system this nonmodal logic is restricted to full propositional logic. Due to the finite state nature of the programs considered, however, every first order formula can at least theoretically be expressed by a propositional formula.

The properties can be classified as safety and progress properties. Safety properties are the basic *co* properties, and the *unless*, *stable*, *invariant*, and *constant* properties which can be expressed in terms of *co* and boolean connectives (the *invariant* property requires the initial condition of the program as well). Progress properties are the basic *transient* properties, the *ensures* properties which can be expressed in terms of *transient*, *co*, and boolean connectives, and the *leads-to* properties defined as the reflexive, transitive and disjunctive closure of the *ensures* relation (for finite state programs disjunctivity is redundant, since finite disjunctivity can be deduced from the laws for *ensures* and from transitivity).

The model checking algorithm for most kinds of properties can make direct use of the partitioning of the transition relation of a program induced by the statements of that program. As a consequence, in many model checker invocations the entire transition relation never has to be constructed (not even implicitly) increasing the performance of the model checker significantly. All safety properties and all basic progress properties (with the only exception of *leads-to* properties) can be dealt with very efficiently that way. The algorithm for checking *leads-to* properties employs a fixpoint computation based on the predicate transformer *wlt* (see [JKR 89]) taking the fairness constraints on program executions into account.

Each property given as input to the UMC system is connected to exactly one program, that serves as the structure over which the property is to be interpreted (and hence the local variables of which it can refer to). The system Furthermore keeps track of the checking status of each property, namely whether it has not been checked yet, whether it has been proved successfully, whether an attempted proof failed (but still has the potential to succeed with respect to a stronger invariant), or whether it has been proved invalid for the program.

2.4 The Role of Invariants

Strictly speaking the satisfaction of a property for a given program is defined with respect to the set of reachable states of the program. A complete way of checking a property would be to first determine the strongest invariant of the program, and then check the property with respect to that strongest invariant. Unfortunately computing the strongest invariant is often very complex, even if techniques like iterative squaring are used. In particular intermediate results of the computation can become too big to be dealt with efficiently, even though the final result may be reasonably small. On the other hand it is often the case that it is not necessary to known the strongest invariant of a program, a weaker invariant may already suffice to proof other properties. Sometimes even the trivial type invariant (asserting only that all variables take on values from their respective types) does the job, allowing a 'proof from the program text'.

The UMC system deals with this situation by conceptually associating an invariant lattice with each property. Such a lattice is determined by the program providing the context for the property. Lattice elements are invariants of the program ordered by logical implication. The top (weakest) element of such a lattice is the type invariant of the program, the bottom (strongest) element is the strongest invariant of the program. The invariant lattice of a program is complete, for a finite state program it is finite. Furthermore each invariant in the lattice associated with a property is labeled indicating whether the property has been proved with respect to the invariant, whether a proof has failed, or whether the result is still unknown.

The checking status of a property is uniquely determined by the labeling of the invariant lattice, e.g. a property is proved to be not valid if and only if the strongest invariant is labeled with 'fail'. By keeping track of strongest invariants for which a proof failed, and weakest invariants for which a proof succeeded (provided they exist), the model checker can manage the checking of a property and at times even determine, whether a checking attempt succeeds or fails without actually invoking the model checking algorithm.

In the current system the user has some limited flexibility in choosing invariants with respect to which a property is to be checked. The system maintains up to three invariants for each program the user can choose from: the first is the type invariant (corresponding to directly checking), the second is the strongest invariant (provided it has been computed successfully), and the third is the so called current invariant being the conjunction of all successfully proved invariants of the program. By choosing the order of model checker invocations on invariant properties the user can control the current invariant being built.

For computing the strongest invariant of a given program the UMC system furthermore provides three algorithms the user can choose from: forward chaining, forward chaining with frontier nodes, and iterative squaring. The forward chaining algorithms compute the strongest invariant as the set of reachable states starting with the initial set of states and iteratively adding the (right)

image under the transition relation until a fixpoint is reached. In the ordinary forward chaining algorithm, the image of all collected node is computed in each step, whereas the frontier node algorithm forward chaining computes the image of only the states that were added in the previous step. Both algorithms require a number of iterations equal to the greatest distance of a state from an initial state (i.e. in the order of the diameter of the state graph). Ordinary forward chaining requires fewer OBDD operations, but is more likely to produce bigger intermediate results. On the other hand the iterative squaring algorithm computes the transitive closure of the transition relation by iterative squaring and from this the strongest invariant as the (right) image of the set of initial states under the closure relation. It requires only a number of iterations in the order of the logarithm or the diameter of the reachable state graph, unfortunately in most cases it produces intermediate results that are to big to be handled practically.

2.5 Debugging Information

In addition to a simple indication of whether a check succeeded or not, it is crucial for the debugging of programs that the system provide additional diagnostic information that helps to track down a problem. Currently the UMC system is capable of providing the user with some such diagnostic information including witnesses for violated conditions (counterexamples).

For properties that require a simple implication to hold (*co*, *invariant*) a witness state violating the implication is reported. For all properties that requiring some kind of stability or continuation test on statements to succeed (all but *leads-to* properties), a violating statement is indicated as well as a witness state and its successor under the given statement violating the required condition are reported. Properties that require some basic progress in form of helpful transitions (*transient*, *ensures*) cause the system to report such helpful transitions (or the lack thereof). Finally diagnostic output for *leads-to* properties is produced containing information about the number of fixpoint iterations performed, and about whether a fixpoint had been reached, as well as a witness state from which there is a fair unfulfilled computation is reported (in case the property is not proved valid).

2.6 Aspects of Using OBDDs

Although the input language and the compiler for it make the underlying representation of programs and properties by OBDDs transparent to the user, there are some important considerations to follow with respect to system performance. The strong dependence of the size of OBDDs depending on the variable ordering used is well known. In the current revision of the UMC system the user has a very limited way of influencing that ordering, namely by rearranging the variable declarations in the declare section of programs. All bits needed to represent one

variable correspond to consecutive OBDD indices, the order in which variables are mapped to increasing indices coincides with the order they appear in the declare section. The next revision of the system will provide a more flexible way of assigning indices to variable bits, some heuristics about promising orderings need to be implemented as well.

In order to allow the user to evaluate the performance of the system and to compare the efficiency of different operations, a range of performance indicators is provided. These indicators make it possible to monitor the memory usage, the efficiency and load of various hash tables and caches, and the sizes of OBDDs representing computed results.

3 Some Examples

Before we give a detailed description of the input language of the UMC system, we want to present a few simple examples that illustrate several aspects of what can be done with the system, show a few sample inputs and hint at some strategies to be tried out when facing the task of proving a property for a given program.

3.1 Simple – Properties and Invariants

Let us first consider a trivial program that still gives us our first example of the syntax of the input language, shows all different kinds of properties and illustrates the use of different invariants:

```
program Simple
  declare
    var x,y: boolean;
  initially
    x;
    y;
  assign
    x,y := true, x;
end;

x co x;
constant true;
x ensures y;
y unless x;
stable x;
invariant x /\ y;
true --> y;
```

```
y co y;
y co x;
constant x;
invariant y;

transient y;
```

The program *Simple* declares two boolean variables x and y, defines as the initial state the state $(x = true, y = true)$, and describes the transition relation by stating that the value of x in a successor state be *true*, and the value of y in a successor state is the same as the value of x in the current state. Clearly together with the initial condition this amounts to the program never leaving the initial state.

After having parsed the program and the properties and checking all properties directly (i.e. with respect to the type invariant which asserts that x and y are boolean variables and hence either *true* or *false*), we find that the last five properties are not proved. For example the property `constant x;` fails because the stability of $x =$*false* is violated by the assignment.

Of course we know, that this situation can never arise in a program execution since $x \wedge y$ is an invariant of the program (which is confirmed by the model checker). But we need to put this knowledge to use in order to be able to prove some properties. By proving that $x \wedge y$ is an invariant, that invariant is conjoined to the current invariant of the program. If we now check the last five properties with respect to the current invariant rather with respect to only the type invariant, we find that only the very last property fails to be proved.

Concerning this not yet proved property `transient y;` we have to find out whether there is a still stronger invariant which respect to which it could be proved. Of course $x \wedge y$ is the strongest invariant of the program since it characterizes the singleton set of reachable states. Computing the strongest invariant with the model checker makes this knowledge available to the system, which immediately labels the property as failed without a further check.

3.2 Cycle – Finite Range Arithmetic

In our second example we illustrate the use of finite range integers and point at some consequences of the definition of finite state arithmetic. The following program declares a finite range type of 8 elements and two variables of that type. Only variable z is actually used in the program, variable n appears later in some properties where it is used to express some form of universal quantification.

```
program Cycle
  declare
    type R = [0..7];
    var z, n: R;
```

```
  initially
    true;
  assign
    z := z+1;
end;

z=5 --> z=4;

z=n --> z>n;
z=n /\ z<7 --> z>n;
```

The initial state of the program is left arbitrary, in any program execution 1 is repeatedly added to z. Arithmetic on finite range integers is essentially performed modulo the number of elements in the range. Therefore in the above program adding 1 to z is done modulo 8, resulting in a cyclic structure of the range. It is clear from this that the first property is satisfied (adding 1 to 5 seven times indeed yields 4), which is confirmed by checking that property directly.

Unbounded behavior of programs clearly cannot be expressed in a finite state setting, some specifications such as 'z is always increased' do simply not hold. Such a property could be expressed in general UNITY logic by using universally quantified variables. The above property could be expresses as $\forall n :: z = n \mapsto z > n$. We can achieve something similar in our finite state setting by declaring an additional variable n in the program, that, however, is not referred to or even modified in the program itself. As a result there are no constrains on the value of that variable, allowing it to take on any value when it appears in a property. Such a property can only be proved valid, if it does not depend on n, i.e. if it holds for all values of n. In that sense the second property of our example expresses the fact that z always increases, which of course does not hold. (Note that there is no way of defining a non-trivial linear order on a cyclic group that is monotonic with respect to addition. Since an order relation is still a useful construct in many specifications, we decided to define a linear order by injecting the cyclic group into the integers and then using the standard at-most ($\leq$) relation on integers.)

If we take the finite state nature and with it the boundary case ($z = 7$) into account and eliminate it from the property, we obtain the third property listed, which then is easily proved.

3.3 Mutex – A Mutual Exclusion Algorithm

As a last example we show a somewhat more complex program implementing a mutual exclusion algorithm for two processes. This example is taken from [Mis 90]:

```
program Mutex
  declare
    type PC = (noncritical, requesting, trying, critical, exiting);
    var m, n: PC;
    var u, v, p: boolean;
    var hu, hv: boolean;
  initially
    ~u;
    ~v;
    m = noncritical;
    n = noncritical;
  assign
    u, m := true, requesting   if hu /\ m = noncritical;
    p, m := v, trying          if m = requesting;
    m    := critical           if ~p /\ m = trying;
    u, m := false, exiting     if m = critical;
    p, m := true, noncritical  if m = exiting;

    v, n := true, requesting   if hv /\ n = noncritical;
    p, n := ~u, trying         if n = requesting;
    n    := critical           if p /\ n = trying;
    v, n := false, exiting     if n = critical;
    p, n := false, noncritical if n = exiting;

    hu   := ~hu;
    hv   := ~hv;
end;

invariant ~(m = critical /\ n = critical);
invariant u == (m >= requesting /\ m <= critical);
invariant m = critical \/ m = exiting ==> ~p;
invariant v == (n >= requesting /\ n <= critical);
invariant n = critical \/ n = exiting ==> p;
invariant (u == (m >= requesting /\ m <= critical))
          /\ (m = critical \/ m = exiting ==> ~p);
invariant (v == (n >= requesting /\ n <= critical))
          /\ (n = critical \/ n = exiting ==> p);
invariant (u == (m >= requesting /\ m <= critical))
          /\ (m = critical ==> ~p);
invariant (v == (n >= requesting /\ n <= critical))
          /\ (n = critical ==> p);

m = trying unless m = critical;
m = requesting --> (p == v) /\ m = trying;
```

```
m = critical --> p;
m = requesting --> m = critical;
```

Variables m and n serve as program counter in the two processes, boolean variables u, v, and p are used to encode operations of a shared queue between the two processes. Variables hu and hv finally are used to model the nondeterministic behavior of the two processes with respect to their remaining in their noncritical sections. For a more detailed discussion of the program and of its properties we refer to the above mentioned paper.

The first and the last property are of particular interest, as they express the mutual exclusion and the absence of starvation properties. Not surprisingly neither of these properties can be checked directly, but after having proved the the second, the fourth, the eighth and the ninth listed invariants, all properties with the exception of the third, fifth, sixth and seventh invariants can be proved by invoking the model checker with respect to the current invariant. The remaining four invariants do not hold, this example was the first application of the UMC system in which errors in a manual proof were discovered.

4 The Input Language

After the examples of inputs to the UMC system given in the previous section we now proceed to a more rigorous and complete description of the input language. This section can be used as a reference for the abstract syntax and the semantics of the input language, although our treatment of the semantics is informal. For a summary of the concrete syntax we refer the reader to appendix A. A discussion of the notation used for the syntax definitions can be found there as well.

Our presentation of the input language consists of four parts. First we state the general form of an input to the UMC system. We then introduce the pervasive construct of expressions and the type system employed for defining them. The presentation of the structure and the meaning of programs and the various kinds of properties completes the language description.

4.1 Input Structure

The input to the UMC system consists of a sequence of input units being either programs or properties according to the following rules:

input ::= *unit**

unit ::= *program* ;
| *property* ;
| `in` *name* `:` *property* ;

Each program unit introduces a new global name (viz. the program name, see below), it is an error to have programs with same names occurring in an input file. Each property unit is associated with exactly one program, in the first simple form of a property unit given above the property is associated with the most recent preceding program in the input file, in the second form the property is associated with the program with the given name. It follows that the order of the units in the input file matters, furthermore it is an error to refer to a program name before the program has been defined.

4.2 Expressions and Types

The benefits of using type systems in programming are well known. In addition to the ability of recognizing certain program errors already at compile-time, in the context of program verification and model checking in particular a suitable type system provides the user with a set of invariants that are derivable purely from the program text by simple syntactic operations. These invariants come free for the programmer in that she has no proof obligations in order to establish them.

We therefore adopt a simple strong type system of finite types for the UMC input language. In this type system every expression occurring in a program or in a property has a statically well defined type. In the following we introduce the available types, the operations defined for elements of the various types, and the rules for correctly typing expressions.

Types There are five different kinds of types in the UMC type system. These are boolean, finite range integer, enumeration, program, and unspecified integer type. Type equivalence is structural equivalence of the type definitions.

Expressions of boolean type can take on one of two values denoted by the globally predefined boolean constants `true` and `false`. Operations on boolean expressions include the boolean connectives *and*(/\), *or*(\/), *implies* (==>), *follows from*(<==), *equivalence* (==), *antiequivalence* (=|=), and *negation* (~). Furthermore the result of applying any of the relational operators (see below) to suitable expressions is of boolean type. The expressions making up the initially sections of programs, the guards of assignments, and all properties with the exception of the constant properties are required to have boolean type.

A finite range integer type is determined by two integers, the lower bound and the upper bound of allowed values (with both boundary values included). The range of allowed values must not be empty. Operations on finite range integer types are the arithmetic operators *unary plus* (+), *unary minus (negation)*(-), binary *addition* (+) and *subtraction* (-), and the relational operators *equals* (=), *unequals* (~=), *greater*(>), *at least* (>=), *less* (<), and *at most* (<=). The type rules for the arithmetic operations are explained below. For the relational operations it is required that both operands be of exactly the same type, the resulting expression is of boolean type. The order relation for a finite range

integer type is the projection of the order relation on the integers to the finite range of the type.

Enumeration types are determined by an ordered nonempty sequence of constants of that type. A type definition of an enumeration type (see declare section below) defines the type and a set of constants of that type. This makes it illegal to use the same constant name in different enumeration type definitions (in the same scope), since this would result in a multiply defined name. Only the relational operations (as listed above) are defined for enumeration types, the underlying ordering is the ordering of the constant names in the type definition. The two operands in a relational operation must be of the same type, the resulting expression is of boolean type.

Every program in the input text defines a global constant of program type, bearing the name of the program. As of now there are no operations defined for type program, however, program constants are used in defining the context for properties.

Finally there is an unspecified integer type, although it is rarely used. Only number literals (like `53`) and expressions built from only number literals and arithmetic operations (so called number literal expressions, which have a static value) possibly can have that type, and only if they are not in a context in which a (specified) finite range type can be inferred for them. The rules for inferring a finite range type of a number literal expression are as follows: first every number literal expression is replaced by the uniquely statically determined number literal or boolean obtained by evaluating the expression over the integers. Any number literal still occurring as operand in a relational expression then has the same type as the other operand of that expression (for the value conversion of out of range values see below). Only number literals appearing as subexpressions in a binary arithmetic expression or as expression of a *constant* property retain the unspecified integer type.

Arithmetic In defining the semantics of arithmetic operations on finite range integer types it was our design decision to avoid the introduction of undefined expressions that would complicate the language both on the syntactic and on the semantic level, and to keep many laws we know from integer arithmetic valid. These goals have been achieved at the price of losing some other laws (especially concerning ordering) and of obtaining a very restrictive type system. We believe, however, that this should not cause any problems to the user, as long as she is aware of the fact that she is dealing with finite range integers. On the contrary, we make the limitations explicit rather than implicitly relying on some inflexible assumption about how arithmetic is performed (e.g. such as treating all integer arithmetic modulo the wordsize of the underlying machine).

A range type `[l..h]` contains $h - l + 1$ elements and is augmented with the group structure isomorphic to the cyclic group of $n = h - l + 1$ elements C_n. The unit element is the unique number u in the range such that $u \equiv 0 \bmod n$.

Two numbers a of type `[la..ha]` and b of type `[lb..hb]` can be added, if and only if their types ranges contain the same number of elements, i.e. if and only if $na = nb$ with $na = ha - la + 1$ and $nb = hb - lb + 1$. The type of their sum $a + b$ then is `[la+lb..ha+lb]`, its value is the unique number s in that range such that $a + b \equiv s \bmod na$.

The negation of a has type `[-ha..-la]`, its value is the unique number i in that range such that $i + a \equiv 0 \bmod na$. The difference of a and b is defined by $a - b = a + (-b)$, it therefore has type `[la-hb..ha-hb]`.

Adding a constant expression with unspecified integer type (see above) and value v to an expression of type `[l..h]` results in an expression of type `[l..h]`, the value of which is obtained by adding 1 to the typed value a total of v times (or subtracting 1 a total of $-v$ times if v is negative). Subtraction as usual is addition of the negated second argument. Finally assigning a constant expression with unspecified integer type and value v to a variable of type `[l..h]` results in taking the variable on the unique value u in the type range such that $u \equiv v \bmod (h - l + 1)$.

With these definitions the commutativity and associativity as well as the cancellation laws hold, furthermore all arithmetic expression that are well typed are well defined.

Since relational operations on finite range integer types are defined with respect to the linear ordering of the integers, however, laws connecting arithmetic operations with order relations do not hold in general. For example the expression $x < x + 1$ is not necessarily true: if x has type `[3..7]` and value 7, $x + 1$ evaluates to 3, and $7 < 3$ is false.

It is worth mentioning that such violations all have to do with crossing the boundaries of the finite range type, they can be avoided implicitly by choosing the range such that the boundary values are not reached, or by explicitly guarding against crossing them, e.g. by using the assignment `x := x + 1 if x<7` rather than `x := x+1` for variable x of type `[3..7]`.

The typing rules may seem complicated in general, but they simplify considerably when dealing only with finite range types with a lower bound of 0: the type `[0..h]` defines the cyclic group C_{h+1}, and the two most frequently used operations addition of variables and addition of subtraction of a variables and a constant become ordinary group operations.

Scoping The name space of a collection of programs and properties consists of two levels: global names can be referred to in all programs and properties, local names can be referred to only in the program they are defined in and the properties of that program. Each name refers to either a type, a variable or a constant. Names must be unique within the scope they are defined in, local names hide global names if they are textually the same.

There are a few predefined global names (see below). The only other global names denote constants of type program, they are defined as the program names

whenever a program is defined.

Local names are defined in the declare section (see below) of a program.

Predefined Symbols There are three predefined global names as listed in the following table:

Name	Kind	Remark
`boolean`	type	
`true`	constant	of type `boolean`
`false`	constant	of type `boolean`

It is possible to locally hide these names by redefining them in the declare section of a program.

4.3 Programs

A program consists of a program name and three sections as follows:

program ::= `program` *name declare initial assign* `end`

The program name defines a global variable of type program. The *declare* section introduces a local name scope for the program and allows the definition of types, variables, and constants in it. The *initial* section defines the set of initial states for the program, and the *assign* section characterizes the transition relation of the program expressed in the form of guarded assignments. Each of these sections must be present but can be empty.

Declare Section The declare section consists of a sequence of type, variable, or constant declarations:

declare ::= `declare` (*ditem* `;`)*

ditem ::= `var` *name*+, `:` *type*
| `const` *name*+, `:` *type* `:=` *expression*
| `type` *name*+, `=` *type*

Each such declaration introduces new local names (namely the names in the list of names), defines them as being either types, variables or constants, states their type, and – for constants – defines their value; All the names in such a declaration are defined simultaneously, they all share the same types and – for constants – the same value. Constants have to be typed. The expression given in a constant definition must be a constant expression, i.e. it must not refer to any variables; furthermore its type must be equivalent to the type of the constant (where an unspecified integer type is equivalent to any finite range integer type).

The names introduced in a declaration are added to the name space before the next declaration is processed, thus allowing to use local names in defining other local names.

Type equivalence is determined by structural equivalence, in particular all the types mentioned in a type declaration are equivalent. Type expressions denoting types can take on one of three forms:

type ::= *name*
| `[` *expression* `..` *expression* `]`
| `(` *name**, `)`

In the first form the type of a previously defined type is denoted. The second form denotes the finite range integer type with the bounds determined by the given expressions in the following way: both expression, which must be constant expressions, are evaluated according to the arithmetic laws stated earlier, the unique results are then regarded as integers defining the bounds of the type. Furthermore the lower bound must not exceed the upper bound. As an example consider the declarations

```
const n: [0..4] := 1;
type R = [9+n .. 2];
```

Since the type of `9+n` is `[0..4]` the value of `9+n` is 0, resulting in the type `R = [0..2]`.

The third form denotes an enumeration type and implicitly defines a set of constants of that type (viz. the names listed between parentheses). Two enumeration types are equivalent if they are made from the same constants in the same scope.

The declare section defines the state space of a program as the cartesian product of the domains of all variables declared.

Initially Section The initially section consists of a (possibly empty) list of expressions:

initial ::= `initially` (*expression* `;`)*

The expressions are required to have boolean type, their conjunction defines the predicate that characterizes the set of initial states of the program. An empty initially section corresponds to the predicate true, i.e. to the entire syntactic state space.

Assign Section The assign section consists of a possibly empty set of possibly guarded multiple assignments:

assign ::= `assign` ($name^+$, `:=` *rhs* `;`)*

rhs ::= $expression^+$,
| $case^+$,

case ::= $expression^+$, `if` *expression*

Each assignment specifies a non empty list of variables on the lefthand side of the assignment operator. All names in that list have to be distinct. There are two forms of the righthand side *rhs*: the first corresponds to an assignment with only one case and guard true, the second allows to list several cases with arbitrary boolean guards. The number of righthand side expressions must be equal to the number of lefthand side variables both in the first form and in each case of the second form. Furthermore the types of corresponding variables and expressions must be equivalent. There is an additional restriction on the use of guards in different cases: in every state of the state space in which two guards of different cases of the same assignment are true, all corresponding righthand side expressions must be equal. This makes an assignment such as

```
x := ~x if y,
     y if true;
```

for boolean variables `x, y` illegal, since in the state (x = true, y = true) both guards are true, but the righthand side expressions for x are different (false, true).

The transition relation defined by the program includes the pair (s,t) of states of the state space, if and only if there is an assignment case of one of the assignments, whose guard evaluates to true in s and whose multiple assignment execution updates s to t. In particular the empty assign section corresponds to the empty transition relation.

4.4 Properties

Any property is defined in the context of exactly one program, which is either named explicitly or is the most recently defined program in the input text (see above). As a consequence all global symbols and all local symbols defined in the associated program can be referred to in the property. The syntax for all eight unconditional properties of UNITY logic is as follows:

property ::= `constant` *expression*
| `invariant` *expression*

| `stable` *expression*
| *expression* `co` *expression*
| *expression* `ensures` *expression*
| *expression* `-->` *expression*
| *expression* `unless` *expression*

With the exception of the constant properties all expressions are required to be of boolean type, whereas for constant properties any type but program type is allowed.

5 Discussion and Outlook

The UMC system is designed to be a building block of a verification and design support system. In order to be of practical value to a designer of distributed programs it must provide an efficient and complete verification method, support for managing programs and specifications, and means for debugging faulty programs. It moreover should be easy to use both with respect to the logic employed and to the user interface.

These criteria are met by the UMC system in the following ways: it utilizes a OBDD based symbolic representation of programs and specifications, it is based on UNITY logic, its input language is very close to the UNITY notation, and it provides means for managing and debugging programs and properties via a graphical user interface.

Thus the UMC system can be considered at least a potentially valuable tool for the designer of distributed systems, as it allows her to verify properties of her programs while lowering the risk of introducing errors a more often than not tedious manual proof might carry with it. Moreover the interactive investigation and utilization of various invariants allows the user to put her understanding of the program to use for the verification of more difficult to prove properties.

Although the current revision of the UMC system deals with the issues related to model checking of single finite-state programs in a very satisfactory way, our current work suggests several improvements to increase the system performance and to extend the range of verification problems that can be dealt with effectively. These extensions concern the data types, the exploitation of program composition and the parameterization of programs.

In order to increase the expressiveness of the input language we are planning to add structured data types such as records, arrays, and finite functions, that have been proved to be extremely useful in programming, as well as to rework the type definition of finite range integers to better meet the requirements of given problems (e.g. when using a finite range integer type user may really ask for a cyclic group, a bounded segment of the integers or a scalar set without ordering among the elements).

Program composition can be achieved both on a syntactic and on a structural level. With syntactic composition the input language provides features for writing a program as the composition of several constituent parts, internally the program is then represented in its entirety as far as model checker invocations are concerned. This level of compositionality is extremely useful for designing bigger programs, especially if syntactic means for hiding local variables and for combining program components are provided. However, this approach is likely to suffer from the state explosion problem when the product state space needs to be built. In our research we therefore focus on structural composition, which treats components of programs separatly and attempts to check their composition by reasoning about properties of the components. This approach requires techniques beyond mere model checking, in particular validity of inference checking and compositional reasoning based on the rely/guarantee paradigm seem to be needed.

The purpose of allowing parameterized programs is to represent families of structures in a concise way, and to verify possibly unboundedly many structures in finite time. A careful use of decision procedures based on inductive methods will be required to extend the applicability of a model checker to certain classes of parameterized structures. We are currently investigating how such classes can be characterized, how known techniques from automated theorem proving can be employed, and how our model checking algorithms can be extended to deal with these classes.

A Grammar

We summarize the syntax of the input language for the UMC system using an extended BNF. Nonterminals are written in *italics*, terminals in `typewriter type style`. The standard metasymbols used are ::= to separate the left- and righthand sides of grammar rules, | to separate alternative righthand sides, parentheses (and) for grouping, the * operator indicating zero or more repetitions of the construct it is applied to and the + operator indicating one or more repetitions of the construct it is applied to. Furthermore there are two additional postfix operators $*_{,}$ and $+_{,}$ the first indicating zero or more repetitions of the construct it is applied to, with successive repetitions being separated by commas, the second similarly indicating one or more comma separated repetitions.

The language of allowable inputs is generated by the nonterminal *input*, the following grammar rules, and some additional semantic restrictions that are outlined in the section on the input language.

input ::= *unit**

unit ::= *program* ;

| *property* ;
| `in` *name* `:` *property* ;

property ::= `constant` *expression*
| `invariant` *expression*
| `stable` *expression*
| `transient` *expression*
| *expression* `co` *expression*
| *expression* `ensures` *expression*
| *expression* `-->` *expression*
| *expression* `unless` *expression*

program ::= `program` *name declare initial assign* `end`

declare ::= `declare` (*ditem* ;)*

ditem ::= `var` *name*+, `:` *type*
| `const` *name*+, `:` *type* `:=` *expression*
| `type` *name*+, `=` *type*

type ::= *name*
| `[` *expression* `..` *expression* `]`
| `(` *name*+, `)`

initial ::= `initially` (*expression* ;)*

assign ::= `assign` (*name*+, `:=` *rhs* ;)*

rhs ::= *expression*+,
| *case*+,

case ::= *expression*+, `if` *expression*

expression ::= *name*
| *number*
| `(` *expression* `)`
| `+` *expression*
| `-` *expression*
| *expression* `+` *expression*
| *expression* `-` *expression*
| *expression* `=` *expression*
| *expression* `~=` *expression*
| *expression* `>` *expression*
| *expression* `<` *expression*

| *expression* `>=` *expression*
| *expression* `<=` *expression*
| *expression* `/\` *expression*
| *expression* `\/` *expression*
| *expression* `==>` *expression*
| *expression* `<==` *expression*
| *expression* `==` *expression*
| *expression* `=|=` *expression*
| `~` *expression*

The input language is case sensitive. The input is made up from a sequence of tokens and white space (i.e. blanks, tabs, and newline characters). White space only is significant for separating tokens, beyond that it is ignored. Tokens are either terminals as appearing in the grammar rules above, or strings of characters representing the *name* and *number* nonterminals.

A *name* is any string of one or more characters made up from letters and digits starting with a letter. A *number* is a string of one or more digits. A *name* must not be one of the terminal keywords mentioned in the grammar rules above, a *number* is restricted in its size depending on the implementation of the system.

For easy reference we list all keywords of the input language in alphabetic order: `assign`, `co`, `const`, `constant`, `declare`, `end`, `ensures`, `if`, `in`, `initially`, `invariant`, `program`, `stable`, `transient`, `type`, `unless`, and `var`.

Similarly we summarize all operators and their binding powers by listing them in increasing order of binding power (operators on the same line share the same binding power). All operators are left associative as far as typing allows:

`==` (boolean equivalence), `=|=` (boolean antiequivalence, exclusive or)
`==>`, `<==` (boolean implication, both directions)
`/\` (boolean conjunction), `\/` (boolean disjunction)
`~` (boolean negation)
`=`, `~=`, `>`, `>=`, `<`, `<=` (nonboolean relational operators)
`+` (addition), `-` (subtraction)
`-` (unary arithmetic negation)

Finally we list the remaining terminal tokens of the input language which are used as delimiters: `:=`, `(`, `)`, `[`, `]`, `;`, `:`, `,`, `..`, and one special token denoting the *leads-to* operator: `-->`.

B Implementation Details

The UMC system started out in 1992 as an experimental model checker for UNITY written in Scheme on a Macintosh computer. After a first prototype

had shown the feasibility of the approach, the system was rewritten in C++ for the UNIX environment. The current revision (1.10) is implemented using the GNU development tools on a SUN SPARCStation under the X Window System with a user interface based on Motif. The lexical analyzer and the parser for the UMC input language were built with the help of the GNU scanner and parser generators flex and bison.

The internal representation of programs and properties and the actual model checking algorithms are based on our own implementation of an ROBDD package (Reduced Ordered BDDs, many ideas for an efficient implementation are taken from [BBR 90]). This allows us to tailor the representation details (in particular as far as housekeeping and gathering statistical information is concerned) to the specific needs of the UMC system. A thorough understanding of the details of the representation might become even more important in the future when different data structures (e.g. inductive BDDs ([GF 93])) might supplement OBDDs.

References

[BBR 90] K. S. Brace, R. E. Bryant, R. L. Rudell, *Efficient Implementation of a BDD package*, in Proceedings of the 27th ACM/IEEE Design Automation Conference 1990.

[CM 88] K. M. Chandy, J. Misra, *Parallel Program Design: A Foundation*, Addison Wesley 1988.

[GF 93] A. Gupta, A. L. Fisher, *Parametric Circuit Representation Using Inductive Boolean Functions*, in Proceedings of the Conference on Computer Aided Verification 1993, Springer LNCS 697.

[JKR 89] C. S. Jutla, E. Knapp, J. R. Rao, *A Predicate Transformer Approach to Semantics of parallel Programs*, PODC 1989.

[Mis 90] J. Misra, *A Family of 2-process Mutual Exclusion Algorithms*, Notes on UNITY, 13.

[Mis 93] J. Misra, *Safety, Progress*, unpublished manuscripts, August 1993.