MUNITY

H. Peter Gumm *

Abstract

MUNITY is a program verification system for parallel programs expressed in the UNITY language of K.M.Chandy and J.Misra [1]. Given a program and a list of hypothesized properties expressed in the temporal UNITY logic, MUNITY generates and largely proves a list of verification conditions. The development environment supports the interactive discovery and proof of relevant UNITY properties expressing stability and progress and, if appropriate, also points to bugs in the specification of the parallel program. Dealing with data types abstractly we show how data type requirements can be easily discovered by the system.

1 Introduction

UNITY is a theory for specifying and verifying parallel programs. It consists of a simple programming language together with a temporal logic style program logic and a proof system. UNITY programs are specified independent of architectural concerns as a set of nondeterministically executable actions. Mapping a UNITY program to a particular parallel architecture is considered separately. The UNITY logic is based on the temporal connectives *unless* and *ensures*. Other temporal operators such as *stable*, *until* and *leadsTo* are derived from these two connectives. Hoare triples {P} s {Q} tie the operational semantics to the programming logic. The temporal properties are defined using quantification over all assignments s in a UNITY program of Hoare triples {P} s {Q} .

MUNITY is a program verification system based on the UNITY calculus. Its purpose is to semiautomatically prove that a given UNITY program will satisfy certain relevant temporal properties. More than just checking truth or falsehood, we expect that the system is designed to interactively guide the user to the discovery of relevant program properties and their proofs.

We will start with a brief introduction to UNITY and refer the reader to the original book by K. M. Chandy and J. Misra [1] for a detailed exposition.

*SIEMENS AG., ZFE IS INF 2, Otto Hahn Ring 6, 8000 München 83, Germany and Fachgebiet Informatik, Philipps Universität, 3550 Marburg

1.1 UNITY programs and their semantics

The basic actions of UNITY are expressed using *enumerated assignments*. In its simplest form, an enumerated assignment is just a multiple assignment of a list of terms to a corresponding list of variables. In general, the right hand side of the multiple assignment may consist of several guards, each one guarding its own list of terms, so that an enumerated assignment has the general form

```
<EnumAssign> ::=
        <variableList> :=      <termList> [ [ IF <guard> ]
                             [ ´~´ <termList> IF <guard> ]*     ]
```

If more than one guard is true, then all the corresponding term lists are required to yield the same list of values.

Assignment statements are constructed as parallel compositions

$$A_1 \parallel \ldots \parallel A_n$$

of assignments A_i. Such compositions can be expressed more generally using *quantification* over a tuple of variables satisfying a Boolean expression

$$\langle \parallel i_1, ...i_n : < \texttt{boolExpr} > :: A_{i_1,\ldots,i_n} \rangle$$

A UNITY program is a *set* of assignment statements. These statements are connected using the *nondeterministic choice* operator "[]". Again there is a quantified version, called a *quantified statement list* :

$$\langle [] i_1, \ldots, i_n : < \texttt{boolExpr} > :: A_{i_1,\ldots,i_n} \rangle$$

A UNITY program has up to four sections. In a *declare-section* variables and their types are declared, in an *always section* variables are defined as a function of other variables. These defined variables cannot be assigned to and are best compared to macros. Initial conditions are declared in the *initially-section* and finally the *assign-section*, consisting of a set of UNITY assignments represents the executable part of a UNITY program.

The semantics of a UNITY program specifies that starting from a state where the condition given in the initially-section is true, statements are then repeatedly selected from the assign-section nondeterministically and executed. For the nondeterministic selection a simple *fairness condition* is assumed: Each statement is selected infinitely often. [1]

A UNITY program never terminates, although it may reach a fixed point. A fixed point is a state which remains unchanged for every statement. A logical description of a fixed point can be easily derived from the source code using the following idea : Each assignment `x := t IF b` contributes the condition `b`

[1] It is entirely possible that in a conditional statement '`x := t IF b`' the condition `b` happens to be false each time the statement is selected.

`==> x=t`. The fixed point FP is then characterized by the conjunction of all these conditions.

Although the expressive power of UNITY seems to be somewhat weak at first glance, the ability to treat arbitrary data structures on the right hand side of an assignment statement permits one to hide complicated calculations that are not relevant in the interplay with other components. Furthermore, the virtual absence of any control at the statement level gives the implementor sufficient freedom when mapping a UNITY program to a particular architecture. The following example shows a UNITY specification of a "Faulty Channel" which may lose or duplicate data, but which will not transmit incorrect data. For the implementation we use two queues, `inQ` and `outQ` with operations `IsEmptyQ, head, tail,` and `enQ`, the latter being written as infix ";".

```
Program faultyChannel
declare
  inQ, outQ : queue of data
assign
     inQ, outQ := tail(inQ), (outQ;head(inQ))
                       IF not IsEmpty(inQ)    %  correct transfer
 []  inQ   := tail(inQ)
                       IF not IsEmpty(inQ)    %  data loss
 []  outQ  := (outQ;head(inQ))
                       IF not IsEmpty(inQ)    %  data duplication
end
```

1.2 Program structuring

Structuring of UNITY programs is achieved by decomposing the problem into logical units, implementing each unit as a UNITY program, then expressing the final solution as a union of these smaller programs. A program for the "alternating bit protocol", for instance, could be decomposed into modules representing "sender", "receiver" and two instances of a (parameterized) "faultyChannel" :

```
Program altBit =
   sender
[] receiver
[] faultyChannel(sendChannel)
[] faultyChannel(ackChannel)
```

1.3 The UNITY logic

Properties of UNITY programs can be formulated in a temporal logic language powerful enough to express the relevant aspects of parallel program behavior, such as *safety* and *progress*. Although UNITY programs never terminate, we can still express termination of a particular algorithm as a progress condition:

A program terminates iff eventually it reaches a state satisfying the fixed point condition.

The most basic logical connectives used in MUNITY are called *unless* and *ensures*. Both are defined using a quantification over Hoare triples where the quantification extends over all statements `s` in the assign section A of the program. For predicates P and Q define

$$P \textbf{ unless } Q \leftrightarrow \forall s \in A : \{P \wedge \neg Q\} s \{P \vee Q\}$$

Intuitively, if P is true in some state then it will remain true unless Q is or becomes true. The definition of Hoare triples {P} s {Q} is standard, see e.g. [2]. If `s` is a multiple assignment of the form $\texttt{x}_1 \ldots \texttt{x}_n := \texttt{t}_1 \ldots \texttt{t}_n$ `if b`, then {P} s {Q} translates into the *verification condition*

$$(P \wedge b \Rightarrow Q[x_1, \ldots, x_n / t_1, \ldots, t_n]) \wedge (P \wedge \neg b \Rightarrow Q).$$

This is easily extended to multiple assignments with several guards and to their parallel composition, i.e. to arbitrary assignment-statements.

The *ensures* connective requires that there exists at least one statement in A guaranteeing progress from P to Q, and that no other statement can prevent this progress :

$$P \textbf{ ensures } Q \leftrightarrow (P \textit{ unless } Q) \wedge (\exists s \in U : \{P \wedge \neg Q\} s \{Q\})$$

At the user level there are two operators that are most important to formulate properties of UNITY specifications : *invariant*, to express safety and *leadsTo* (denoted by $\mapsto$), to express progress. If I is the initial condition then

$$\textbf{invariant } P \leftrightarrow (I \Rightarrow P) \wedge \textit{stable } P.$$

The $\mapsto$ relation is defined as the smallest transitive relation containing the *ensures* relation and closed under the disjunction law :

$$\frac{\langle \forall m : m \in W :: P(m) \mapsto Q \rangle}{\langle \exists m : m \in W :: P(m) \rangle \mapsto Q}$$

that is, if $P(m)$ *leadsTo* Q individually for each $m \in W$ then we are allowed to conclude that the disjunction of all $P(m)$ *leadsTo* Q. This law allows us to prove progress conditions by (possibly infinite) case analyses. Interestingly, the finite version of the disjunction law, i.e.

$$\frac{P_1 \mapsto Q, P_2 \mapsto Q}{P_1 \vee P_2 \mapsto Q}$$

is already true for the transitive closure of the *ensures* relation.

2 MUNITY

MUNITY is a verification system for UNITY programs that we have created at the Munich research center of Siemens AG. MUNITY provides a comfortable program development environment, together with an integrated program verifier and theorem prover. The user specifies a program together with external safety and progress conditions formulated in the UNITY logic. The system will formally verify that these conditions are indeed true in the given program. MUNITY supports the incremental development of a UNITY program satisfying given logical properties, or, given a program, the development of proofs for basic UNITY properties. The latter capability has been demonstrated in a case study where we investigated safety and progress properties in a program controlling magnetic levitation trains [3].

2.1 MUNITY specifications

The syntax of a MUNITY program is slightly simpler than the original UNITY syntax. Mainly, parallel composition of assignments has been removed. Since we still have multiple assignments, parallel composition is just extra syntactic sugaring. Also, when inconsistencies between a program and its proposed properties are detected, parallel composition makes it rather difficult for the user to pinpoint an exact source of the problem. In all example programs we have dealt with, equivalent reformulation of parallel composition in terms of multiple assignments has not seriously affected readability.

A MUNITY specification consists of a program name, an *initially-section*, an *assign-section* and an *invariant section*. The initially-section, $True$ by default, contains a boolean condition that is assumed to hold at the start of execution of the program.

The assign-section consists of a set of multiple guarded simultaneous assignments, separated by the "[]"-symbol. In an optional *invariant-section* an *external invariant* for the program may be specified. The concepts of internal and external invariants will be explained later. The full syntax of a MUNITY specification is :

```
<MUNITY-spec> ::= program <id>
                    [ initially  <boolExpr>   ]
                      assign     <statements>
                    [ invariant  <boolExpr>   ]
                  end

<statements> ::= statement [ [] statement ]*

<statement>  ::= <multiple guarded simultaneous assignment>
```

2.2 MUNITY scripts

Along with a MUNITY program we usually want to specify properties of the program, using connectives from UNITY logic, such as *invariant*, *leadsTo*, *unless*, *ensures*, etc. Such properties can be added to the MUNITY specification in a *properties section*. The combination of a MUNITY program and a properties-section is called a *script*.

In the current version, MUNITY can handle the following types of properties: *stable, invariant, unless, ensures*. *LeadsTo* formulas can only be formally verified if they are presented as a transitive chain of *ensures* formulas. Since "leadsTo" is defined by inference rules, full support for the verification of arbitrary temporal logic formulas will only be provided by a proof checker for UNITY logic, whose implementation is in progress.

2.3 External Invariants and the substitution rule

If Q is a property true in every reachable state of program U then Q may implicitly be used as an assumption whenever we reason about program U. This is the substance of the *substitution rule* of [1]. A problem arises, however, when we combine programs U_1 and U_2. Properties true in both components need not remain true in the combination. As an example consider the following two programs:

```
Program U1
initially
  x=y=0
assign
  x := x+1
end
```

```
Program U2
initially
  x=y=0
assign
  y := y+1
end
```

For the first program we find *invariant* $y = 0$, so the substitution rule of [1] allows to infer *invariant* $x = 0 \vee y = 0$ even though $x = 0 \vee y = 0$ is not stable in U1. We obtain the same invariant for U2, but clearly $x = 0 \vee y = 0$ will not remain true in $U_1[]U_2$.

Rather than dropping the substitution rule altogether, we introduce the concept of an *external invariant*. This must be a property *stable* in all component programs and initially true in their composition. The external invariant is the only property which we ever allow to be used as an implicit axiom, i.e. substituting *true*. As seen from each component program, its external invariant is both a promise to and an expectation of the environment. In essence then a UNITY program becomes a pair (U, I) and two programs (U_1, I_1) and (U_2, I_2) can only be composed if $I_1 = I_2$. The aim must therefore be, to find one external invariant that is as strong as possible, yet still is stable in all components. In the above example not more than $x \geq 0 \wedge y \geq 0$ could be used as an external invariant, if the programs are to be combined.

The problem with the original substitution axiom in [1] has been noticed before in the literature. Sanders [4], in fact, has reworked part of the UNITY calculus using UNITY programs indexed with invariants and thus removed an unsoundness of the original calculus.

2.4 Verification conditions

Given a MUNITY script, the verifier has to check correctness of the MUNITY specification, and of the MUNITY properties. Correctness of the specification involves the verification that the proposed external invariant is indeed an invariant of the program part, that consists of the initially- and the assign-sections. Correctness of the MUNITY-properties means that, assuming the external invariant, the claimed properties are true in the UNITY program. The command "inv" checks the external invariant and the command "prove" proves the proposed properties. In either case a syntactical check will be made and errors will be indicated by a message and by a cursor being placed at the offending position in the source code.

When checking invariants or other properties, MUNITY reduces them to their definitions via Hoare triples and generates a corresponding series of implications of first order predicate logic, so called *verification conditions*. One after another they are displayed on the screen and the internal rewrite system attempts a proof. In case a proof does not fully succeed the verification condition is reduced to an equivalent but simpler one and displayed under the header `- Remains to prove -` `!`. The MUNITY script has been verified if all arising verification conditions have been either proven, or the parts that ``remain to prove´´ have been accepted as tautologies by the user.

All verification conditions which the system cannot prove by itself, are simplified and collected in a file `axioms.log`. They can later be passed to a general first order theorem prover. It is important that during proof development the user can concentrate on the design and properties at the UNITY level, without being required to fill in the often tedious details of proofs for predicate logic tautologies, which many times are obvious to humans. Once the development phase is completed, of course, the final MUNITY script cannot be considered formally verified until all propositions from the `axioms.log`-file have been formally verified, using any first order theorem prover.

2.5 Environment and user interface

Currently there are two MUNITY versions available that differ only in the user interface, but not in functionality: a command line version running on SUN workstations and a fullscreen version running on IBM compatible PC's. The latter version is embedded in an *integrated development environment*, quite similar to commercial language implementations, such as, e.g. Turbo-Pascal. In a main editor window the MUNITY-script can be edited and from a menu bar commands such as "Inv", "Prove", "Files", "Options", "Help" can be chosen.

Some commands in turn result in pulldown menus offering more detailed choices according to context. Syntax errors cause the cursor to be placed at the offending spot in the editor window. Verification conditions are displayed in an output-subwindow that opens above part of the editor window. While stepping through the verification conditions, the user may switch back and forth between editor window and output window in order to associate verification conditions with the place in the source code from where they arise.

The command line version requires only an ASCII terminal, commands are entered from the keyboard and syntax errors are indicated by displaying several lines of the source file surrounding the error location and graphically indicating the exact spot of the error.

The PC version is a standalone executable file created from a C and Turbo-Prolog source code, whereas the SUN version requires a PROLOG runtime system - currently we use "Sepia" Prolog [5]. The performance of either systems has never been of much concern, since typically parsing, generation of verification conditions, and proofs or simplifications of individual verification conditions take place in fractions of a second.

3 Interacting with MUNITY

In this section we shall demonstrate with an example how interactive development and proof of a MUNITY-script proceeds. Since the connection between a UNITY program and its properties is established via Hoare triples, one has to start out with (a preliminary version of) a UNITY program. Our development methodology calls for starting out with an abstract version of the program, then refining it step by step, mainly through a refinement of the data structures involved. This methodology follows [1] and was analogously used in describing a train control system in [3] and recently by A. Scholz in verifying the "Sliding Window Protocol".

3.1 Example: Sender-receiver communication

The following UNITY implementation of a communication through unsecure channels is based on the treatment in [1]. To keep the example concise, we shall not explicitly implement faulty channels, rather we have sender and receiver behave in a way that the same message may be sent, received and acknowledged arbitrarily often. Send- and receive-channels are simply modelled by shared variables.

In this first level of abstraction we consider only the ordinal numbers of messages, not their content. Thus the sender sends the sequence of natural numbers and the receiver uses the last number received for an acknowledgement.

```
program altBit1
{declare
  x,       -- send-channel (sender writes, receiver reads)
```

```
  y,       -- receive channel (receiver writes, sender reads)
  ks,      -- number to transmit
  kr       -- last received number
         : integer }
initially
  x=1 and y = 0 and ks = 1 and kr = 0
assign
     ks := y+1          % (1)  prepare next message
  [] x  :=  ks          % (2)  send message (again)
  [] kr := x            % (3)  receive message (again)
  [] y  := kr           % (4)  acknowledge message (again)
end
```

The most important safety and progress conditions will have to state that

- a message (number) received must have been sent
- every number is eventually received (hence has been sent)

In UNITY logic we can express them as

- `invariant kr` $\leq$ `ks`
- `kr = N` $\mapsto$ `kr = N+1`

As a MUNITY convention, identifiers starting with upper case letters denote constants, so any write access is prevented by the parser. Constants occurring within MUNITY properties are implicitly universally quantified. In particular, the second property must be read as

$$\forall N : kr = N \mapsto kr = N + 1$$

.

3.1.1 Safety

Starting with the safety condition, we add the proposed invariant

```
invariant kr <= ks
```

We would like to use it as an external invariant, since we suspect that this condition will be needed to prove the proposed progress condition. The verifier now generates a series of verification conditions of which there are two (caused by program statements 1 and 3) that it cannot prove:

`kr<=ks ==> kr<=y+1` and `kr<=ks ==> x<=ks`

Verification conditions are purely logical statements, so identifiers occurring in verification conditions denote logical variables. Hence the actual proof obligations are :

$\forall kr, ks, y : kr \leq ks \Rightarrow kr \leq y + 1$ and $\forall kr, ks, x : kr \leq ks \Rightarrow x \leq ks$.

Since none of these are true, our proposed invariant is not really invariant. Although *true* in every program state it does not have the self-perpetuating property required of an invariant. Since the verification conditions arise from statements no. 1 and 3, the reason is easily found in the program source: statement 1 changes `ks` without regard to `kr` and similarly, statement 3 changes `kr` without regard to `ks`. Obviously, we have to strengthen the invariant to somehow also involve `x` and `y`. From the previous trials we can read off the conditions `kr <= ks and kr <= y+1 and x <= ks`. Still this turns out not to be an invariant and from the verification conditions that `remain to prove` we are soon led to

`y <= kr <= x <= ks <= y+1`.

More than just stating `kr <= ks` the new invariant guarantees that the values of `kr` and `ks` never differ by more than 1, a crucial property, indeed.

The forced strengthening of the invariant is in complete analogy to what can be experienced in inductive proofs when the inductive hypothesis needs to be strengthened in order to prove some weaker property, or what happens during the proof of a sequential program when the loop invariant needs to be strengthened even though a weaker consequence is required at the loop's exit.

Finding a "good" invariant is always the first and often the most difficult step in proving a UNITY program. The invariant shrinks the space of states to consider, or, viewed differently, provides an extra axiom for further proofs.

3.1.2 Progress

Recalling that $\mapsto$ contains the transitive hull of *ensures*, we show that

$$\texttt{kr} = \texttt{N} \mapsto \texttt{kr} = \texttt{N} + 1$$

using a chain of `ensures`-steps. Such a chain is best discovered by moving backwards from the desired final state.

There are two degrees of freedom here: Since we must achieve a chain

$$\texttt{kr} = \texttt{N} \equiv P_0 \; \textit{ensures} \; P_1 \; ... \; P_{n-1} \; \textit{ensures} \; P_n \equiv \texttt{kr} = \texttt{N} + 1$$

we have to find a proper predicate logical statement P_{n-1} and further, since every ensures step involves an existential quantification over all statements `s` in the program, we must find an appropriate `s`. The latter question is solved most easily: If proper progress is made towards `kr=N+1`, then it can only be through a statement that modifies `kr`, in this case only statement 3 qualifies. MUNITY allows us to annotate ensures conditions with the number of the statement that in the user's opinion will make the progress.

Next, an appropriate P_{n-1} is required. It must be some predicate depending at most on the variables contained in the program, so we formulate the property

$$\texttt{P(x,y,kr,ks) ensures} < 3 > \; \texttt{kr} = \texttt{N} + 1$$

Of the five generated verification conditions, all but one are proved by the system. The only one remaining involves the unknown or *abstract* predicate `P(x,y,kr,ks)`:

```
P(x,y,kr,ks) AND kr<>N+1
AND y<=kr AND kr<=x AND x <= ks AND ks <= y+1
  ==>
x=N+1
```

Since none of the premises allows us to conclude much about the proposed conclusion, the only valid information could be in `P(x,y,kr,ks)`. Since we are still free to choose, we simply put

$$P_{n-1} \equiv \mathtt{P(x,y,kr,ks)} \equiv \mathtt{x} = \mathtt{N} + 1.$$

The fresh attempt to prove `x=N+1 ensures<3> kr=N+1` leaves only one verification condition unproven :

```
kr<>N+1
AND y<=kr AND kr<=N+1 AND N+1<=ks AND ks<=y+1
  ==>
ks=N+1 OR kr=N+1
```

As before, we see how the external invariant appears as an additional axiom in every verification condition. The current one, is indeed a tautology provided that all identifiers denote integers, so in this case the first ensures condition has been proven. We may also interpret it in a different way :

Since we declined to declare variables, this verification condition introduces a data structure constraint for the common type T of `x,y,ks,` *and* `kr`*:*

$$\forall p, q \in T : p \leq q \leq p + 1 \Longrightarrow (q = p) \vee (q = p + 1).$$

After thus establishing $P_{n-1} \equiv \mathtt{x} = \mathtt{N} + 1$ we keep working backwards until we have produced the ensures chain

```
kr=N ensures N=y ensures ks=N+1 ensures x=N+1 ensures kr=N+1,
```

witnessing `kr=N` $\mapsto$ `kr=N+1`. In each case a verification condition quite similar to the last one remains, but no new constraints are introduced.

3.2 Inferring data structures

In the next version the sender sends from an infinite sequence `ms` of messages. Again, `ks` is the count of the current message to be transmitted. The message itself is `ms[ks]`. The send-channel now has two components, index and value which we name `x.dex` and `x.val`. Again we acknowledge with the ordinal number of the last message received. The messages are stored by the receiver in a

queue `mr`. It is not necessary to have specific data types available to implement the queue operations. We simply invent appropriate names for the operations such as `enQ` or `firstQ`, etc. and we are prepared to assume appropriate axioms. Since all UNITY actions must be expressed as assignments, we denote an enqueuing of a new data item `x` into queue `mr` by `mr := enQ(mr,x)`. Our program refines to :

```
Program altBit2
initially
  x.dex = 1 and y = 0 and ks = 1 and kr = 0 and
  mr = Null and x.val = ms[1]
assign
        y           := kr
  []    ks          := y+1
  []    x.dex,x.val := ks,ms[ks]
  []    kr,mr       := x.dex, enQ(mr,x.val)   if kr <> x.dex
invariant
  y <= kr and kr <= x.dex and x.dex <= ks and ks <= y+1
end
```

The external invariant essentially remains, but at first glance it is not clear how to express safety and progress conditions, since our syntax only allows open formulas inside the MUNITY-properties. The safety property must express that *what has been received has also been sent.* Let us call this relation `Prefix`. Specifying the invariant

```
invariant Prefix(mr,ms)
```

we obtain a verification condition

```
Prefix(mr,ms) AND y<=kr AND ... ==> Prefix(enQ(mr,x.val),ms)
```

where the "..." represent the external invariant. Since `x.val` does not occur anywhere in the premise, we must establish some relationships between the variables involved. If `x.val` was just enqueued, we should expect that it was the last message sent, so it ought to have been read off the array `ms` at place `ks`, so the new invariant becomes :

```
Prefix(mr,ms) AND ms[x.dex]=x.val.
```

This produces a verification condition

```
Prefix(mr,ms) AND ms[x.dex]=x.val AND ...
  ==>
Prefix(enQ(mr,x.val),ms)
```

Hence it becomes clear that we must relate `x.dex` to the length of `mr`, so finally we arrive at the invariant:

```
Prefix(mr,ms) and x.val=ms[x.dex] AND kr=length(mr).
```

Since this is the first time that MUNITY ever encounters "length" and "Prefix", we are finished, if all verification conditions can be proven using only such axioms about "Prefix" and "length" that we are willing to assume about these notions, i.e. axioms that are logically consistent. Indeed the following verification is generated:

```
Prefix(mr,ms)
AND ms[x.dex]=x.val
AND kr=length(mr)
AND kr <> x.dex
AND y <= kr <= x.dex <= ks <= y+1
  ==>
Prefix(enQ(mr,ms[x.dex]),ms)
AND x.dex=length(enQ(mr,ms[x.dex])
```

The hypothesis implies that `x.dex=length(mr)+1`, so the following two axioms are needed about `Prefix` and `length` :

```
For all queues q and all arrays s :
   length(Null)=0
   length(enQ(q,x)) = length(q)+1

   Prefix(Null,s)
   Prefix(enQ(q,s[length(q)+1]),s)
```

Assuming these axioms about `Prefix, Null,` and `enQ` we have shown that the queue of messages received is always a prefix of the sequence of messages to be sent, and additionally we have interactively inferred the necessary data structure axioms from the program. We get by without using quantifiers in our specification. The function `length` in fact plays an auxiliary role as a Skolem-function. A progress property is now easily formulated as

$$\texttt{length(mr)} = \texttt{N} \mapsto \texttt{length(mr)} = \texttt{N} + 1$$

Its proof proceeds via the following *ensures*-chain:

```
length(mr)=N    ensures<1> length(mr)=N and kr <> y+1
                ensures<3> length(mr)=N and kr <> x.dex
                ensures<4> length(mr)=N+1
```

In the final version which, due to space limitations, will not be discussed here, only one bit, ks mod 2, is added to the message sent and only one bit, kr mod 2 is acknowledged. It is not hard to adapt the invariant accordingly in order to prove the same safety and progress properties as before.

4 Conclusion

We have described MUNITY, an interactive verifier for UNITY programs and we have demonstrated with an example how such a tool can guide the user to relevant properties of concurrent programs. Proof attempts lead to first order properties from which axioms for the required data types can be inferred.

References

[1] K. M. Chandy, J. Misra *Parallel Program Design.* Addison Wesley, 1988.

[2] J. W. deBakker *Mathematical Theory of Program Correctness* Prentice Hall, Englewood Cliffs, N. J., 1980.

[3] H. P. Gumm *Verifying the Transrapid System in Munity.* Preprint, 1992.

[4] B. Sanders *Eliminating the Substitution Axiom from UNITY Logic.* Formal Aspects of Computing **3**(1991), 189-205. Preprint, 1992.

[5] *Sepia 3.1 User Manual* ECRC, 1991.