A UNITY-based Algorithm Design Assistant*

Michel Charpentier† Abdellah El Hadri‡
Gérard Padiou†

†ENSEEIHT-IRIT
2, rue Charles Camichel
31071 TOULOUSE cedex
e-mail: {charpov, padiou}@enseeiht.fr

‡EMI
Avenue Ibn Sina BP 765
RABAT (MAROC)
e-mail: elhadri@emi.ma

Abstract

We address the problem of the automatic verification of reactive systems. For such algorithms, parallelism, non-determinism and distribution, lead to frequent design flaws and make debugging difficult. Proving programs with respect to their specification may solve both these problems. In this framework, we describe the implementation of an algorithm design assistant based upon the `UNITY` formalism. A theorem prover and a Presburger formulas calculator are used to perform the underlying proofs. We illustrate the main difficulties encountered with representative examples.

Key words: Program verification, reactive programs, `UNITY` formalism, parallelism, distribution, theorem proving.

I Introduction

Concurrency and distribution generate two further difficulties with respect to sequential programming. Concurrency leads to a drastic increase in program states and distribution results in a knowledge loss of both any global state and time. Therefore, program debugging becomes especially difficult.

To cope with these problems, we need to improve and formalize the methods and tools to design such programs. In this framework, two approaches may

*Workshop on Tools and Algorithms for the Construction and Analysis of Systems, May 1995, pages 131–145.

be investigated. Firstly, the development process itself may be assisted and supervised rigorously. Several projects are currently devoted to such computer assisted software engineering (*CASE*) tools. Secondly, the development operations may be improved too. In this case, the use of well formalized methods should enable to produce correct programs more quickly. However, without assistance, proofs are long, tedious and error prone. Therefore, computer assisted proofs appear a worthwhile requirement. We investigate this approach.

Stepwise refinement programming is a well known method. The development of a program consists of a sequence of pairs $(Spec, Prog)$ in which $Spec$ is a specification and $Prog$ its associated implementation. Starting from a high level abstract version, successive refinements generate new pairs to lead to a final program validated with respect to its specification. Each refinement must have well defined semantics in order to check its validity in the context of a specific development. The handling of both the specification and the implementation of a program helps to grasp the problem insofar as these representations complement each other.

However, a parallel calculus model is required to experiment such an approach. This model should:

- provide both a specification and programming language with a well formalized foundation;
- allow a stepwise refinement design with the same formalism all along the development process;
- include parallel features.

According to these requirements, we have chosen the `UNITY` model from Chandy and Misra [CM88] as an experimental basis. Its specification language allows to describe the safety properties and liveness properties of programs and the programming language provides both synchronous and asynchronous statements to express parallelism. Furthermore, this model is general enough to carry out a stepwise refinement approach.

In this framework, the formalization of a development involves two tasks:

- the research and the definition of generic refinements;
- the consistency verification of the pair $(Spec, Prog)$. More precisely, the program $Prog$ must be validated with respect to its specification $Spec$.

About the first task, the refinement method allows to prove only the initial pair $(Spec, Prog)$ if the further performed refinements are sound. Although this point will not be further considered in this paper, we actually search how to refine a centralized algorithm so that it can be efficiently mapped to a target distributed architecture.

The second task raises an automatic proof problem. We study this question in the following sections more precisely and its implementation in a `UNITY` based environment.

II The UNITY formalism

The `UNITY` formalism is based on fair transition systems. It consists of a programming language to describe the state transitions and a specification language based upon linear temporal logic. We only outline the main features. Readers familiar with `UNITY` may skip this section.

II.1 UNITY programs

A `UNITY` program consists of four sections:

- the **declare** section defines a set of typed variables.
- the **initially** section assigns initial values to the variables. A variable may be initialised at most once with an expression of its type. Non-initialized variables have arbitrary values at the beginning of the program execution.
- the **always** section defines some variables as functions of others.
- lastly, the **assign** section contains the program statements. More precisely, a set of multiple-assignment statements are listed. The symbol $[\!]$ is used to separate these assignments.

A program execution runs forever starting from any state satisfying the initial conditions. At any step, a statement is arbitrarily selected for execution. However, each statement is executed infinitely often (weak fairness hypothesis).

Terminating programs are interpreted as programs that reach a fixed point, namely, a stable state. Any further execution step leaves the (stable) state unchanged.

The only `UNITY` statement is the assignment with several forms: conditional, quantified and enumerated. For instance, a conditional multiple assignment has the following syntax:

$$\begin{array}{rcll} v_1, v_2, \cdots, v_n & := & e_1^1, e_2^1, \cdots, e_n^1 & if \;\; g_1 \\ & \sim & e_1^2, e_2^2, \cdots, e_n^2 & if \;\; g_2 \\ & \cdots & \cdots & if \;\; \cdots \\ & \sim & e_1^p, e_2^p, \cdots, e_n^p & if \;\; g_p \end{array}$$

Unlike Dijkstra's guarded commands, when several conditions are simultaneously true, the assigned values must be the same. Each assignment is deterministic. The assignment leaves the variable values unchanged if none of the conditions is true.

Enumerated assignments are a syntactic feature to cut long multiple assignments. For instance, $x := a \parallel y, z := b, c \parallel t := d$ is an equivalent notation for $x, y, z, t := a, b, c, d$.

Quantification is allowed, especially with parallel or exclusive assignments. For instance, the quantified assignment $\langle \parallel i : 1 \leq i \leq N :: A[i] := B[i] \rangle$ assigns the array B to the array A.

II.2 UNITY logic

A specification language is associated to the former operational language. It allows to specify safety and liveness properties as logical expressions. The program variables are operands of such predicates. These properties may be proved with respect to the program component by means of a set of theorems.

The UNITY logic is based on Hoare triples $\{P\}\ s\ \{Q\}$ [Hoa69] in which P and Q are predicates and s a statement. Such a triple has the following interpretation: if the statement s is performed from a program state satisfying P and terminates, the resulting state will verify the predicate Q.

Safety property syntax

Three basic relations allow to express safety properties, namely: **Unless**, **Stable** and **Invariant**. For a program $Prog$ and predicates P and Q, we have the following definitions:

$$P\ \textbf{Unless}\ Q \ \equiv\ \langle \forall s : s\ in\ Prog :: \{P \wedge \neg Q\} s \{P \vee Q\}\rangle$$

(if P holds at some point, P remains true at least until Q holds.)

$$\textbf{Stable}\ \ P \ \equiv\ P\ \ \textbf{Unless}\ \ false$$

(if P becomes true, it remains true forever.)

$$\textbf{Invariant}\ \ P \ \equiv\ (initial\ \ conditions \Rightarrow P) \wedge (\textbf{Stable}\ \ P)$$

(P is true at the program start and all over the execution.)

Liveness property syntax

The basic liveness relation is **ensures** with the following definition: for a program $Prog$ and predicates P and Q

$$P\ \ \textbf{Ensures}\ \ Q \ \equiv\ (P\ \ \textbf{Unless}\ \ Q) \wedge \langle \exists s : s\ \textbf{in}\ Prog :: \{P \wedge \neg Q\} s \{Q\}\rangle$$

(if P holds at some point, P remains true at least until Q becomes true and there exists one statement to eventually validate Q.)

This property allows to define a more powerful relation $\mapsto$ (**leads-to**). $P \mapsto Q$ means that if P holds at some point, Q eventually becomes true.

II.3 Example

We consider a program with two counters i and j. Their behaviour depends on each other.

$$\begin{array}{lll}
\textbf{initially} & i,j \ =\ 0,10; & \\
\textbf{assign} & & \\
 & i := i+1 & \\
[\!] & j := i & \textbf{if}\ \ i \neq 5
\end{array}$$

This program verifies the following safety and liveness properties:

$\forall k$ **Stable** $i \geq k$		– i cannot decrease
$\forall k$ $i = k$ **Unless** $i = k + 1$		– i can only increase by 1 steps
$\forall k$ $i = k$ **Ensures** $i = k + 1$		– i actually increases by 1 steps
Invariant $i \geq 0$		– i is always non negative
Invariant $i \geq 0 \wedge j \geq 0$		– i and j are always non negative
$\forall k$ $j = k \mapsto j > k$		– j increases indefinitely

Note on invariant and "always true" properties

As defined in `UNITY`, a property I is invariant if it holds in the initial state of the program and it is maintained by any transition of the program (stable).

Thus, from a state where I holds, a transition must lead to a state where I still holds. The only knowledge about the starting state is that I holds. In particular, no assumption is made about the reachability of the state. If there exists a state from which I is not maintained, I is not invariant, even if this state is unreachable.

Therefore, a property may hold for all possible executions of a program and not be invariant (i.e. not be maintained from some unreachable state). For instance, in the example above, we do not have **Invariant** $j \geq 0$ although j is never less than zero for all possible executions of the program. Actually, from the (unreachable) state $(i, j) = (-1, 2)$, the second statement does not maintain $j \geq 0$.

A weaker form of invariants has been introduced that exactly fits the notion of *always true* properties [San91]. It is based on the idea of *strongest invariant*. The *strongest invariant* of a program is the strongest (for the imply relation) invariant property of that program. It is defined as the conjunction of all the invariants of the program. In other words, the *strongest invariant* holds in any reachable state and only in the reachable states. It may be used to characterize the reachable states of a program, and thus avoids the above difficulty. An *always true* property is then a property implied by the *strongest invariant*.

In practice, the main difficulty in proving that a given property is *always true* in a program is to find an invariant strong enough to imply the property, that is to specify precisely enough which are the reachable states. Some tools used in proving programs automatically allow to strengthen a given invariant (see VI.2) and even to compute the *strongest invariant* in particular cases (see VI.1).

III The design assistant

III.1 Description

The assistant appears as a syntax-directed editor with an `X Window System` interface for both the `UNITY` language and logic. It has been generated by the `Grammatech Synthesizer Generator` and provides all the functionalities of the editors created with this tool, described in [RT89].

The system handles a two-part text:

- an algorithm description in the UNITY language,
- a specification for both the safety and liveness properties of this algorithm in the UNITY logic. Each property is labelled with a name.

The goal is to control automatically the program with respect to all the specified properties.

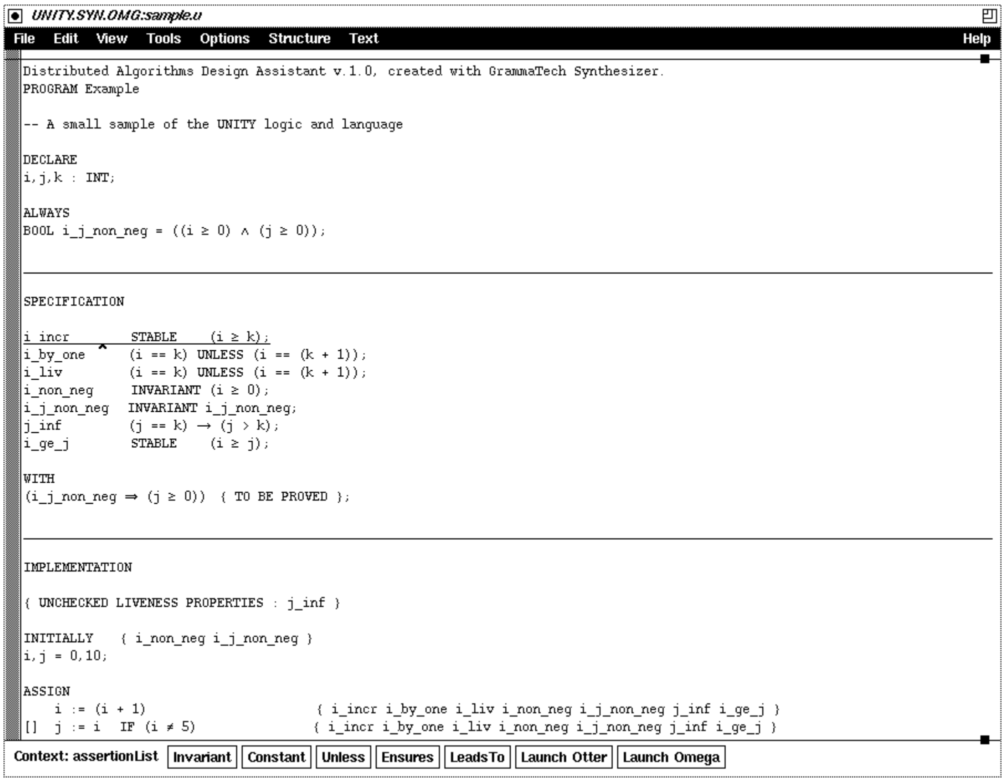

Figure 1: User interface of the assistant

Figure 1 shows the user interface of the environment. The user can edit a syntactically correct text in the UNITY formalism. The editor translates this text into an attributed abstract tree. Incremental analysis is performed by updating attribute values throughout the current tree in response to each editing transaction (text editing, system command, ...).

In this UNITY editor, the attribute evaluation performs the following semantic actions:

- static semantic checking (type control);
- generation of formulas that express the consistency between the specification part and the implementation part;
- calls to a theorem prover to verify the previous formulas. Currently, the user can choose between two tools: the Otter theorem prover (see IV) and the Omega calculator (see V).

The labels of the properties to be checked are displayed after each program statement. In order to check a property, the user must first select this property with the mouse, and then call a theorem prover by clicking on the corresponding button. If the proof of the property succeeds, the corresponding label disappears. For instance, in figure 1, the **Stable** property *i_incr* is selected and its proof may be launched by clicking on a button located at the bottom of the window.

Currently, the environment only manages a subset of the `UNITY` formalism. The language accepts integers and booleans but neither arrays nor statement quantification are allowed. At the logical level, the temporal operator $\mapsto$ is not handled. Such proofs involve too many practical and theoretical difficulties (transitive closure calculus, for example).

In addition to the usual sections of a `UNITY` program (**declare**, **always**, **initially** and **assign**) a **with** section provides a direct interface with the theorem provers. This feature is useful for proving (non temporal) predicates with program variables as operands.

III.2 Examples

The tool has been tested in different areas including distributed systems, hardware design or robotics. We detail three classical synchronisation examples frequently mentioned in related projects.

Peterson's mutual exclusion algorithm

This first example is derived from the *UNITY Verifier* user manual [Kal94a] (Cf VI.1). It consists of an implementation and a specification of the Peterson's algorithm written by J. Misra [Mis90]. This algorithm insures mutual exclusion between two processes. The first five statements represent the first process, and the five next the other process. The two last assignments are used to model the nondeterministic behaviour of the two processes with respect to their remaining in their non critical sections. Variables m and n serve as program counter in the two processes. For instance, the equality $m = critical$ means that the first process is in its critical section.

The specification enumerates eleven safety and liveness properties. The first one (**Invariant** I_ONE) allows us to deduce the most important safety property, namely the fact that $EXCLUSION$ is *always true* in the program. Note that the formula $I\_ONE \Rightarrow EXCLUSION$ is proved through the **with** section.

The assistant provides a TeX translation from its internal text representation. The following program and specification are generated thanks to this translation feature. The provers are unable to check the $three$, $five$, six and $seven$ invariants (labels remain displayed). In fact, these properties do not hold for the program as it was shown by the *UNITY Verifier*. There were errors in the first hand proof.

Program Mutex
declare
m,n : INT;
u,v,p : BOOL;
hu,hv : BOOL

always
INT noncritical $= 0$,
INT requesting $= 1$,
INT trying $= 2$,
INT critical $= 3$,
INT exiting $= 4$,
BOOL EXCLUSION $= \neg((\mathtt{m} = \mathtt{critical}) \wedge (\mathtt{n} = \mathtt{critical}))$,
BOOL I_EIGHT $= ((\mathtt{u} \Leftrightarrow ((\mathtt{m} \geq \mathtt{requesting}) \wedge (\mathtt{m} \leq \mathtt{critical}))) \wedge ((\mathtt{m} = \mathtt{critical}) \Rightarrow \neg \mathtt{p}))$,
BOOL I_NINE $= ((\mathtt{v} \Leftrightarrow ((\mathtt{n} \geq \mathtt{requesting}) \wedge (\mathtt{n} \leq \mathtt{critical}))) \wedge ((\mathtt{n} = \mathtt{critical}) \Rightarrow \mathtt{p}))$,
BOOL I_ONE $= (\mathtt{EXCLUSION} \wedge (\mathtt{I\_EIGHT} \wedge \mathtt{I\_NINE}))$

Specifications

one	**invariant** I_ONE,
two	**invariant** $(\mathtt{u} \Leftrightarrow ((\mathtt{m} \geq \mathtt{requesting}) \wedge (\mathtt{m} \leq \mathtt{critical})))$,
three	**invariant** $(((\mathtt{m} = \mathtt{critical}) \vee (\mathtt{m} = \mathtt{exiting})) \Rightarrow \neg \mathtt{p})$,
four	**invariant** $(\mathtt{v} \Leftrightarrow ((\mathtt{n} \geq \mathtt{requesting}) \wedge (\mathtt{n} \leq \mathtt{critical})))$,
five	**invariant** $(((\mathtt{n} = \mathtt{critical}) \vee (\mathtt{n} = \mathtt{exiting})) \Rightarrow \mathtt{p})$,
six	**invariant** $((\mathtt{u} \Leftrightarrow ((\mathtt{m} \geq \mathtt{requesting}) \wedge (\mathtt{m} \leq \mathtt{critical})))$ $\wedge (((\mathtt{m} = \mathtt{critical}) \vee (\mathtt{m} = \mathtt{exiting})) \Rightarrow \neg \mathtt{p}))$,
seven	**invariant** $((\mathtt{v} \Leftrightarrow ((\mathtt{n} \geq \mathtt{requesting}) \wedge (\mathtt{n} \leq \mathtt{critical})))$ $\wedge (((\mathtt{n} = \mathtt{critical}) \vee (\mathtt{n} = \mathtt{exiting})) \Rightarrow \mathtt{p}))$,
eight	**invariant** $((\mathtt{u} \Leftrightarrow ((\mathtt{m} \geq \mathtt{requesting}) \wedge (\mathtt{m} \leq \mathtt{critical})))$ $\wedge ((\mathtt{m} = \mathtt{critical}) \Rightarrow \neg \mathtt{p}))$,
nine	**invariant** $((\mathtt{v} \Leftrightarrow ((\mathtt{n} \geq \mathtt{requesting}) \wedge (\mathtt{n} \leq \mathtt{critical})))$ $\wedge ((\mathtt{n} = \mathtt{critical}) \Rightarrow \mathtt{p}))$,
ten	$(\mathtt{m} = \mathtt{trying})$ **unless** $(\mathtt{m} = \mathtt{critical})$,
eleven	$(\mathtt{m} = \mathtt{requesting})$ **ensures** $((\mathtt{p} \Leftrightarrow \mathtt{v}) \wedge (\mathtt{m} = \mathtt{trying}))$

with
$(\mathtt{I\_ONE} \Rightarrow \mathtt{EXCLUSION})$

---

implementation

initially
u = false;
v = false;
m = noncritical;
n = noncritical

assign

u,m := true,requesting	**if** $(\mathtt{hu} \wedge (\mathtt{m} = \mathtt{noncritical}))$	
‖		
p,m := v,trying	**if** $(\mathtt{m} = \mathtt{requesting})$	{ five seven }
‖		
m := critical	**if** $(\neg \mathtt{p} \wedge (\mathtt{m} = \mathtt{trying}))$	
‖		
u,m := false,exiting	**if** $(\mathtt{m} = \mathtt{critical})$	
‖		
p,m := true,noncritical	**if** $(\mathtt{m} = \mathtt{exiting})$	
‖		
v,n := true,requesting	**if** $(\mathtt{hv} \wedge (\mathtt{n} = \mathtt{noncritical}))$	
‖		
p,n := ¬u,trying	**if** $(\mathtt{n} = \mathtt{requesting})$	{ three six }
‖		
n := critical	**if** $(\mathtt{p} \wedge (\mathtt{n} = \mathtt{trying}))$	
‖		
v,n := false,exiting	**if** $(\mathtt{n} = \mathtt{critical})$	

```
[]
    p,n := false,noncritical          if (n = exiting)
[]

    hu := ¬hu
[]
    hv := ¬hv

end {Mutex}
```

The parking problem

The parking model [AHV83] illustrates the problem of critical resource allocation to concurrent processes. The parking size C represents the number of available resources. A car arrival event simulates a resource request. An effective car entrance into the parking represents a resource allocation and an exit stands for a resource release.

The specification is based upon event counters respectively representing the number of arrivals A, entrances E and departures D. The basic invariant property expresses that the number of cars inside the parking always remains less then or equal to the parking capacity, that is $E - D + F = C$ where F is the number of free places and E, D and F are non negative.

A distributed implementation can be seen as a parking with several access points. Each access point manages its own local counters of events and free places, namely a_i, e_i, d_i and f_i with the representation invariant:

$$A = \sum_i a_i, \quad E = \sum_i e_i, \quad D = \sum_i d_i, \quad F = \sum_i f_i$$

In the following program, we consider a three access point parking. UNITY assignments update event counters and the different access points exchange free places by means of a virtual ring topology.

```
Program Parking

declare
    C : INT;
    a1,a2,a3,e1,e2,e3,d1,d2,d3,f1,f2,f3 : INT;
    ka,ke,kd : INT

always
    INT D = ((d1 + d2) + d3),
    INT E = ((e1 + e2) + e3),
    INT A = ((a1 + a2) + a3),
    INT F = ((f1 + f2) + f3)
```

```
Specifications
    Inv_gl_Impl     invariant  (((E − D) + F) = C),
    f123_ge_0       invariant  (((f1 ≥ 0) ∧ (f2 ≥ 0)) ∧ (f3 ≥ 0)),
    a123_ge_0       invariant  (((a1 ≥ 0) ∧ (a2 ≥ 0)) ∧ (a3 ≥ 0)),
    e123_ge_0       invariant  (((e1 ≥ 0) ∧ (e2 ≥ 0)) ∧ (e3 ≥ 0)),
    d123_ge_0       invariant  (((d1 ≥ 0) ∧ (d2 ≥ 0)) ∧ (d3 ≥ 0)),
    a1_Croit        (a1 = ka)  unless  (a1 > ka),
    e1_Croit        (e1 = ke)  unless  (e1 > ke),
    d1_Croit        (d1 = kd)  unless  (d1 > kd)
```

with
$(((((E - D) + F) = C) \wedge (F \geq 0)) \Rightarrow ((E - D) \leq C))$

---

implementation

initially
a1,a2,a3,e1,e2,e3,d1,d2,d3 = 0,0,0,0,0,0,0,0,0;
f1,f2,f3,C = 2,2,1,5

assign

	a1 := (a1 + 1)	
‖	e1,f1 := (e1 + 1),(f1 − 1)	**if** ((e1 < a1) ∧ (f1 > 0))
‖	d1,f1 := (d1 + 1),(f1 + 1)	**if** (d1 < e1)
‖	a2 := (a2 + 1)	
‖	e2,f2 := (e2 + 1),(f2 − 1)	**if** ((e2 < a2) ∧ (f2 > 0))
‖	d2,f2 := (d2 + 1),(f2 + 1)	**if** (d2 < e2)
‖	a3 := (a3 + 1)	
‖	e3,f3 := (e3 + 1),(f3 − 1)	**if** ((e3 < a3) ∧ (f3 > 0))
‖	d3,f3 := (d3 + 1),(f3 + 1)	**if** (d3 < e3)
‖	f1,f2 := (f1 − 1),(f2 + 1)	**if** (f1 > 0)
‖	f2,f3 := (f2 − 1),(f3 + 1)	**if** (f2 > 0)
‖	f3,f1 := (f3 − 1),(f1 + 1)	**if** (f3 > 0)

end {Parking}

Mutual exclusion with fault tolerance

We now address the problem of distributed mutual exclusion and we focus our attention on fault tolerance. A well-known answer to this problem is based on token passing. The token is unique and only the process which owns the token can enter the critical section. If the system is logically organized in a ring structure and the token circulates on the virtual ring of the processes, the liveness property asserting that any request sent by a process wanting to enter its critical section is eventually accepted, is guaranteed. However, a fault tolerant solution must handle a token loss. In such a case, this loss must be detected and a new single token must be generated.

The following UNITY specification and program describe such an algorithm and its main progress and safety properties. The system consists of three processes, with identical behaviours, represented by similar assignments (with the process identity as parameter).

A process i asks to enter the critical section by setting s_i_req to true. It actually enters the critical section when s_i_excl becomes true. This can only happen when the process has the token, that is when s_i_tok is true. The process exits the critical section by reseting s_i_req and s_i_excl to false. This behaviour can recur indefinitely. The token circulates on the ring by updating any pair (s_i_tok, s_k_tok). Token existence and singleness correspond to the fact that one and only one variable s_i_tok is true at a time. Moreover, any process can lose the token. One selected process (process 1 here) can decide to generate a new token. It starts an investigation around the ring to be sure there is no token, and then it creates a new one.

```
Program jeton

declare
    s1_req,s2_req,s3_req : BOOL;
    s1_excl,s2_excl,s3_excl : BOOL;
    s1_tok,s2_tok,s3_tok : BOOL;
    s1_invst,s2_invst,s3_invst : BOOL

always
    BOOL TOK_ON_S1 = ((s1_tok ∧ ¬s2_tok) ∧ ¬s3_tok),
    BOOL TOK_ON_S2 = ((s2_tok ∧ ¬s3_tok) ∧ ¬s1_tok),
    BOOL TOK_ON_S3 = ((s3_tok ∧ ¬s1_tok) ∧ ¬s2_tok),
    BOOL TOK_UNIQUE = ((TOK_ON_S1 ∨ TOK_ON_S2) ∨ TOK_ON_S3),
    BOOL TOK_EXISTS = ((s1_tok ∨ s2_tok) ∨ s3_tok),
    BOOL CRITICAL_S1 = (s1_req ∧ (s1_excl ∧ (¬s2_excl ∧ ¬s3_excl))),
    BOOL CRITICAL_S2 = (s2_req ∧ (s2_excl ∧ (¬s1_excl ∧ ¬s3_excl))),
    BOOL CRITICAL_S3 = (s3_req ∧ (s3_excl ∧ (¬s1_excl ∧ ¬s2_excl))),
    BOOL EXCL_FREE = ¬((s1_excl ∨ s2_excl) ∨ s3_excl),
    BOOL EXCLUSION = ((CRITICAL_S1 ∨ CRITICAL_S2) ∨ CRITICAL_S3),
    BOOL WHEN_INVST =        ((s1_invst ⇒ (¬s1_tok ∧ ¬s1_excl)) ∧
                             ((s2_invst ⇒ (¬s2_tok ∧ ¬s2_excl)) ∧
                             (s3_invst ⇒ (¬s3_tok ∧ ¬s3_excl)))),
    BOOL WHEN_TOK = (((s1_tok ⇒ ¬s1_invst) ∧ (s2_tok ⇒ ¬s2_invst)) ∧
                             (s3_tok ⇒ ¬s3_invst)),
    BOOL SINGLE = ((TOK_UNIQUE ∨ ¬TOK_EXISTS) ∧ WHEN_TOK),
    BOOL WHEN_EXCL =         ((s1_excl ⇒ (s1_req ∧ (¬s2_tok ∧ ¬s3_tok))) ∧
                             ((s2_excl ⇒ (s2_req ∧ (¬s1_tok ∧ ¬s3_tok))) ∧
                             ((s3_excl ⇒ (s3_req ∧ (¬s1_tok ∧ ¬s2_tok))) ∧
                             (SINGLE ∧ WHEN_INVST))))
```

```
Specifications
    WHEN_EXCL     invariant  WHEN_EXCL,
    WHEN_INVST    invariant  WHEN_INVST,
    WHEN_TOK      invariant  WHEN_TOK,
    SINGLE        invariant  SINGLE,
    MUTEX         invariant  ((EXCLUSION ∨ EXCL_FREE) ∧ WHEN_EXCL)

with
    (TOK_ON_S1 ⇒ ¬(TOK_ON_S2 ∨ TOK_ON_S3)),
    (TOK_ON_S1 ⇔ (TOK_EXISTS ∧ (¬s2_tok ∧ ¬s3_tok)))
```

```
implementation

initially
    s1_tok,s2_tok,s3_tok = true,false,false;
    s1_req,s2_req,s3_req = false,false,false;
    s1_excl,s2_excl,s3_excl = false,false,false;
    s1_invst,s2_invst,s3_invst = false,false,false

assign
    s1_req := true
[]
    s1_excl := true                               if  (s1_tok ∧ s1_req)
[]
```

```
    s1_req,s1_excl := false,false            if s1_excl
[]
    s1_tok := false

[]
    s2_req := true
[]
    s2_excl := true                          if (s2_tok ∧ s2_req)
[]
    s2_req,s2_excl := false,false            if s2_excl
[]
    s2_tok := false

[]
    s3_req := true
[]
    s3_excl := true                          if (s3_tok ∧ s3_req)
[]
    s3_req,s3_excl := false,false            if s3_excl
[]
    s3_tok := false

[]
    s1_tok,s2_tok := false,true              if (s1_tok ∧ ¬s1_req)
  || s2_invst := false                       if (s1_tok ∧ ¬s1_req)
[]
    s2_tok,s3_tok := false,true              if (s2_tok ∧ ¬s2_req)
  || s3_invst := false                       if (s2_tok ∧ ¬s2_req)
[]
    s3_tok,s1_tok := false,true              if (s3_tok ∧ ¬s3_req)
  || s1_invst := false                       if (s3_tok ∧ ¬s3_req)

[]
    s1_invst := true                         if (¬s1_tok ∧ ¬s1_excl)
[]
    s2_invst := s1_invst                     if (¬s2_tok ∧ (¬s2_excl ∧ s1_invst))
[]
    s3_invst := s2_invst                     if (¬s3_tok ∧ (¬s3_excl ∧ s2_invst))
[]
    s1_tok,s1_invst := true,false            if (s1_invst ∧ (s2_invst ∧ s3_invst))
end {jeton}
```

IV The theorem prover Otter

IV.1 Presentation

By means of attribute grammars, the editor part of the tool performs an attribute evaluation based on *weakest precondition* (*wp*) [Dij76] and generates a set of predicates. These predicates have to hold for the program to respect its specification.

Therefore, we need to use a theorem prover to check the predicates validity. After having studied different tools ([KZ87, GG91, Ast92]), based on rewriting or resolution, we have chosen the `Otter` (*Organized Techniques for Theorem-proving and Effective Research*) theorem prover [McC90, McC91], mainly because of its simplicity (that makes easier the interfacing with the editor), its speed, and its ability to work *entirely* automatically. We emphasize that the formulas to prove automatically are numerous but quite simple.

IV.2 Checking specifications

The UNITY specification language is based on the respect of state formulas at different points of a program (*Hoare-triples*). This kind of logic is very useful, and Dijkstra introduced a useful tool for reasoning about it, known as *weakest precondition* (*wp*) [Dij76]. For a state formula Q and a statement s, $wp\,(s, Q)$ is the *weakest* state formula that must hold *before* s to ensure that Q holds *after* s. Therefore, a state formula P is a sufficient precondition for Q with respect to s if and only if it is a *stronger* formula than $wp\,(s, Q)$, that is:

$$\{P\}\ s\ \{Q\} \quad \textit{iff} \quad P \Rightarrow wp\,(s, Q)$$

The function $Q \to wp\,(s, Q)$, from predicates to predicates, expresses the semantics of the statement s, interpreted as a predicate transformer.

While reasoning about UNITY programs, *wp* is very useful because of its computability. One can automatically compute $wp\,(s, Q)$ and transform a specification problem $\{P\}\ s\ \{Q\}$ into the logical formula $P \Rightarrow wp\,(s, Q)$. This deductive approach can deal with infinite state spaces (in contrast with model checking), as, for example, in the parking program (see III.2).

Application to UNITY:

As we have seen, UNITY only provides a single kind of statement, namely the multiple assignment. Therefore, we only have to define *wp* for such assignments. A formula Q will hold after an assignment s if the formula it comes from through s held before executing s. Thus, the *wp* for a formula Q is obtained by variable substitution in Q:

$$wp\,(v_1, \cdots, v_n := e_1, \cdots, e_n \ ,\ \ Q) \quad \equiv \quad Q|_{e_1, \cdots, e_n}^{v_1, \cdots, v_n}$$

This method still works with guarded assignments. An assignment with n guards can be seen as n different assignments, each one having to fit the specification. The *wp* has to be sufficient whichever assignment is actually performed, but the guard can strengthen the precondition for the selected assignment. Finally, one must consider the *nop* statement in the case where no guard holds (no assignment is performed).

For an assignment s of the form:

$$\begin{array}{rcll}
v_1, v_2, \cdots, v_n & := & e_1^1, e_2^1, \cdots, e_n^1 & if\ \ g_1 \\
 & \sim & e_1^2, e_2^2, \cdots, e_n^2 & if\ \ g_2 \\
 & \cdots & \cdots & if\ \ \cdots \\
 & \sim & e_1^p, e_2^p, \cdots, e_n^p & if\ \ g_p
\end{array}$$

we obtain:

$$\{P\}\ s\ \{Q\}\ \ \textit{iff}\ \ \bigwedge_{i=1}^{p} (P \wedge g_i \Rightarrow wp\,(\alpha_i, Q)) \ \bigwedge\ (P \wedge \neg g_1 \wedge \cdots \wedge \neg g_p \Rightarrow Q) \qquad (1)$$

where α_i is the assignment $v_1, v_2, \cdots, v_n \ := \ e_1^i, e_2^i, \cdots, e_n^i$ and where each $wp\,(\alpha_i, Q)$ is computed by substitution over Q.

In the UNITY logic, a constraint of the form X **Unless** Y requires to check the Hoare triple $\{X \wedge \neg Y\}\ s\ \{X \vee Y\}$ for all assignments s. For a *nop* statement, this boils down to verify $X \wedge \neg Y \Rightarrow X \vee Y$ which is a tautology.

Therefore, for any **Unless** property, the last conjunction of formula 1 needs not be computed, since this part of the formula expresses the *nop* case. This simplification remains possible with the **Stable**, **Constant** and **Invariant** constraints which are special cases of the **Unless** relation.

V The Presburger calculator Omega

V.1 Presentation

Reactive programs, like those we consider, often use integer counters. However, classical theorem provers, like `Otter`, are not well suited to symbolic arithmetic calculations. Therefore, we decided to try another kind of tool.

`Omega` [Pug92, KMP+94] is a calculator for Presburger formulas. It is not a theorem prover, but manipulates on sets and tuple relations based on Presburger arithmetic, a class of logical formulas built from affine constraints over integer variables. Such a formula is of the form:

$$\begin{array}{llcll} & \phi \wedge \psi & \qquad & or & \phi \vee \psi \\ or & \neg\phi & & or & \exists\, v_1, v_2, \cdots, v_n : \phi \\ or & \forall\, v_1, v_2, \cdots, v_n : \phi & & or & C \end{array}$$

where ϕ, ψ are Presburger formulas and C is a logical constraint of the form $e_1, e_2, \cdots, e_n\ \ op\ \ e'_1, e'_2, \cdots, e'_n$ with *op* among $=, \neq, <, >, \leq, \geq$. The e'_i are affine expressions on the variables.

A set is defined as follow:

$$\{[u_1, u_2, \cdots, u_n] : \phi\};$$

where ϕ is a Presburger formula, and a tuple relation is of the form:

$$\{[u_1, u_2, \cdots, u_n] \rightarrow [v_1, v_2, \cdots, v_n] : \phi\};$$

`Omega` then offers a variety of set and relational operators to handle these sets and relations, such as **union**, **subset**, **join**, etc.

V.2 Checking specifications

We have added a second interface to the syntax directed editor that allows us to check UNITY specifications with `Omega`. We translate preconditions and postconditions into sets of state, and the multiple assignment into a tuple relation.

To check an assertion like $\{P\}\ s\ \{Q\}$, one can use the notion of *strongest postcondition* (sp). For a statement s and a predicate P, $sp\,(s, P)$ is the *strongest* state formula that holds after having executed s from a state in which P holds. Therefore, a formula Q is a necessary postcondition of P with respect to s if and only if it is a *weaker* formula than $sp\,(s, P)$, that is:

$$\{P\}\ s\ \{Q\} \ \ \textit{iff}\ \ sp\,(s, P) \Rightarrow Q$$

For an assignment s of the form:

$$\begin{array}{rcl} v_1, v_2, \cdots, v_n & := & e_1^1, e_2^1, \cdots, e_n^1 \;\; if \;\; g_1 \\ & \sim & e_1^2, e_2^2, \cdots, e_n^2 \;\; if \;\; g_2 \\ & \cdots & \cdots \qquad\quad if \;\; \cdots \\ & \sim & e_1^p, e_2^p, \cdots, e_n^p \;\; if \;\; g_p \end{array}$$

we obtain:

$$\{P\}\ s\ \{Q\} \;\; \textit{iff} \;\; \bigwedge_{i=1}^{p} (sp\,(\alpha_i, P \wedge g_i) \Rightarrow Q) \;\bigwedge\; (P \wedge \neg g_1 \wedge \cdots \wedge \neg g_p \Rightarrow Q)$$

where α_i is the assignment $v_1, v_2, \cdots, v_n \;:=\; e_1^i, e_2^i, \cdots, e_n^i$.

Instead of handling a predicate directly, one can consider the set of the states for which the predicate holds. A state is the tuple built from the values of the variables in the program. An assignment is a function from states to states. The formula $sp\,(s, P) \Rightarrow Q$ is handled as the set relation $\mathcal{S}\,(\mathcal{P}) \subset \mathcal{Q}$ where $\mathcal{S}$ is the function corresponding to the statement s and $\mathcal{P}$ and $\mathcal{Q}$ are the sets of states corresponding to the formulas P and Q. This approach avoids to compute the *strongest postcondition* explicitly and allows to compute neither *weakest precondition* nor *strongest postcondition* at the editor level.

VI Related work

We distinguish two main approaches in the program proof domain.

On the one hand, the programmer has to describe a complete and formal proof by hand. Afterwards, this proof is automatically checked by a verification system. In this case, no assistance is provided in generating the proof. Systems like *HOL* [GM93], *Deva* [CJL+89] and *PVS* [SOR93] are used to perform such verifications. The tool does not handle both specification and program, but only the hand proof resulting from such a confrontation. `UNITY` proof systems have been described with *HOL* [And92] and with the *Boyer-Moore logic* [Gol92].

On the other hand, the proof construction from a program and its specification is handled at the tool level. The *TLP* system [Eng94], based on the temporal logic of actions (*TLA*) [Lam94], illustrates this one approach. We now focus on two others projects with a closer relationship to our work. The first one, called *UNITY Verifier* [Kal94a, Kal94b], is also based on the `UNITY` programming notation, but uses model checking to perform the proofs. The second one, called STeP [Ma94], allows to validate a program with respect to a linear temporal logic specification.

VI.1 The UNITY Verifier

Model checking is a traditional way for automatically proving programs. It consists in checking that a model (a program) is correct with respect to a formula (a specification) usually by exhaustively checking all possible transitions.

The *UNITY Verifier* is a model checker intended for interactively verifying concurrent programs written in `UNITY`. Through a `X Window System` interface, it operates on a restricted part of the `UNITY` formalism (no array, no statement quantification, no **always** section) and, due to its model checking algorithm, it only handles finite state programs.

Its main attractive feature is its ability to generate the *strongest invariant* (see II.3) for a finite state program and thus, to directly handle *always true* properties. However, *always true* properties raise compositional issues [Col94] and non finite state programs cannot be handled, especially those with integer variables.

VI.2 The STeP system

The STeP project (*Stanford Temporal Prover*) is a system being developed to support the computer-aided formal verification of concurrent and reactive systems based on fair state transition models and temporal logic specifications. Thus, it handles both a linear temporal logic specification and a program written in a parallel language called *SPL*.

The proofs either can be entirely automatic or can be user directed. The system combines model checking and deductive methods to handle non finite state programs. The programming language *SPL* is more complete and deterministic than `UNITY`, and provides synchronisation facilities (such as semaphores, for instance), but `UNITY` remains sufficient to express parallelism with a high degree of non determinism.

Finally, an interesting feature of the system is its ability to generate invariants from the program text and to strengthen user given invariants.

VII Conclusions

With regard to this experiment, the use of several provers with their own capabilities seems to be a more practical way than looking for a very general tool. The STeP system also adopts this approach. Therefore, we think of using other provers, especially the Boyer and Moore theorem prover [BM88].

Automatic invariant generation (as, for instance, in the *UNITY Verifier* and the STeP system) provides an important proof assistance. However, the *strongest invariant* cannot be generated by computation for non finite state programs. In our approach, refinements may allow a stepwise invariant derivation.

Hundred of proofs were necessary to check the small programs described above. Large programs remain out of reach. In front of this difficulty, we are studying two possibilities:

- investigate modularity. Modules are proved independently and further linked with each other. Consequently, sound compositional operators must be available. `UNITY` provides such operators and allows this research direction.
- define generic sound refinements.

In both cases, the assistant will have to check if the refinements are allowed. We investigate both these directions. Generic refinements have already been defined for mapping a centralized algorithm to a distributed architecture [FMP93].

References

[AHV83] F. André, D. Herman, and J.-P. Verjus. *Synchronisation de programmes parallèles.* Dunod Informatique, 1983.

[And92] F. Andersen. *A Theorem Prover for UNITY in Higher Order Logic.* PhD thesis, Technical University of Denmark, 1992.

[Ast92] O. Astrachan. Exploring Model Elimination Theorem Proving. Technical Report CS-1992-22, Duke University, Durham, ,NC USA, December 1992.

[BM88] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*, volume 23 of *Perspectives in Computing.* Academic Press, 1988.

[CJL+89] J. Cazin, R. Jacquart, M. Lemoine, P. Michel, and P. Maurice. *Method driven programming.* G.X. Ritter Editor, Information Processing 89. Elsevier Science Publishers B.V. (North-Holland), 1989.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[Col94] P. Collette. *Design of Compositional Proof Systems Based on Assumption-Commitment Specifications. Application to UNITY.* Thèse de docteur en sciences appliquées, Faculté des Sciences Appliquées, Université Catholique de Louvain, June 1994.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, 1976.

[Eng94] U. Engberg. *TLP Manual.* Computer Science Department, Aarhus University, Ny Munkegade, Building 540, DK-8000 Aarhus C, DENMARK, release 2.5a edition, May 1994.

[FMP93] M. Filali, Ph. Mauran, and G. Padiou. Raffiner pour répartir. Technical Report 93-29/R, Institut de Recherche en Informatique de Toulouse, France, November 1993.

[GG91] S. J. Garland and J. V. Guttag. A Guide to LP, The Larch Prover. Technical Report 82, Systems Research Center, Digital Equipment Corporation, December 1991.

[GM93] M.J.C. Gordon and T.F. Melham. *Introduction to HOL : A Theorem Proving Environnement for Higher Order Logic.* Cambridge University Press, 1993.

[Gol92] D. M. Goldschlag. *Mechanically Verifying Concurrent Programs.* PhD thesis, University of Texas at Austin, May 1992.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[Kal94a] M. Kaltenbach. *Model Checking for UNITY : The UV System.* Department of Computer Sciences, The University of Texas at Austin, revision 1.10 edition, May 1994.

[Kal94b] M. Kaltenbach. *The UV System. User Interface Manual.* The University of Texas at Austin, revision 1.13 edition, December 1994.

[KMP+94] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. *The Omega Calculator and Library, version 0.90.* Dept. of Computer Science and Institute for Advanced Computer Studies, Univ. of Maryland, College Park, MD 20742, November 1994.

[KZ87] D. Kapur and H. Zhang. *RRL : Rewriting Rule Laboratory User's Manual.* Department of Computer Science, State University of New York at Albany, The University of Iowa, June 1987.

[Lam94] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[Ma94] Z. Manna and al. SteP : The Stanford Temporal Prover. Technical Report STAN-CS-TR-94-1518, Department of Computer Science, June 1994.

[McC90] W. W. McCune. OTTER 2.0 Users Guide. Technical Report ANL-90/9, Mathemetics and Computer Science Division, Argonne National Laboratory, Illinois, March 1990.

[McC91] W. W. McCune. What's New in OTTER 2.2. Technical Report ANL/MCS-TM-153, Mathemetics and Computer Science Division, Argonne National Laboratory, Illinois, July 1991.

[Mis90] J. Misra. A Family of 2-process Mutual Exclusion Algorithms. Notes on UNITY, 13, 1990.

[Pug92] W. Pugh. The Omega Test : a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, August 1992.

[RT89] T.W. Reps and T. Teitelbaum. *The Synthesizer Generator Reference Manual.* Texts and Monographs in Computer Science. Springer-Verlag, 1989.

[San91] B.A. Sanders. Eliminating the Substitution Axiom from UNITY Logic. *Formal Aspects of Computing*, 3(2):189–205, April-June 1991.

[SOR93] N. Shankar, S. Owre, and J.M. Rushby. A tutorial on Specification and Verification Using PVS (Beta Release). Technical report, Computer Science Laboratory, SRI International,, Menlo Park, CA, March 1993.