Correctness of Real Time Systems by Construction*

Jozef Hooman

Dept. of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
e-mail: wsinjh@win.tue.nl

Abstract. To design distributed real-time systems in a top-down way, we present a mixed formalism in which programs and assertional specifications are combined. Specifications consist of an assumption-commitment pair, extending Hoare logic to real-time and progress properties. By defining the theory in the PVS specification language, the interactive proof checker of PVS can be used to reason in this framework. We show how this tool can be used during the design of real-time systems to derive programs that are correct by construction.

1 Introduction

A formal framework for the top-down design of distributed real-time systems is presented. By verifying all design steps during the process of program development, a real-time system is obtained which is correct by construction. This requires a compositional proof method in which the specification of a compound programming construct can be derived from the specification of its components without knowing the implementation of these components.

Inspired by the compositional framework of classical Hoare triples (precondition, program, postcondition) for partial correctness [2], we have developed an assertional method for the specification and verification of real-time systems. The assertion language has been extended with timing primitives and the interpretation of triples has been adapted such that properties of both terminating and nonterminating computations can be verified. To indicate the differences with traditional Hoare logic, we use the words "assumption" and "commitment" instead of, respectively, "precondition" and "postcondition". The resulting framework has been applied to several examples such as a water level monitoring system [4], a distributed real-time arbitration protocol [5], and a chemical batch processing system [6].

In this paper we reformulate this approach slightly to obtain a mixed formalism in which programs and specifications are combined in a unified framework. (Similar to, e.g., the mixed terms of Olderog [8].) In such a framework one can

* Appeared in: Proceedings Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, LNCS 863, Springer-Verlag, pp. 19–40, 1994.

freely mix assertional specifications and constructs from the programming language. This makes it possible to express the intermediate stages during program design and to formalize the process of program design. We extend the work on mixed formalisms to real-time and show that top-down program derivation is also possible for distributed real-time systems.

The application of this formal method to large realistic systems clearly requires some form of mechanical support. For instance, one would like to check proofs mechanically, to construct proofs interactively, and to discharge simple verification conditions automatically. Therefore we report in this paper about the use of the verification system PVS (Prototype Verification System) [9] during top-down design in our assumption-commitment framework. The PVS specification language is a strongly-typed higher-order logic. Specifications can be structured into a hierarchy of parameterized theories. There are a number of built-in theories (e.g., reals, lists, sets, ordering relations, etc.) and a mechanism for automatically generating theories for abstract datatypes. The PVS system contains an interactive proof checker with, for instance, induction rules, automatic rewriting, and decision procedures for arithmetic. Further PVS proof steps can be composed into proof strategies.

We describe how our mixed assertional framework can be defined in PVS and how PVS can be used during the top-down design of distributed real-time systems. Hence we formulate our theory directly in PVS. In Section 2 we start with the definition of the basic framework in PVS, considering only sequential programs. We give the semantics of programs, define specifications, and formulate a refinement relation. Proof rules for these programs are formulated in Section 3 (soundness of these rules has been proved in PVS). We indicate that the standard refinement calculus [7] for non-real-time programs is embedded in our framework, illustrated by a simple example of integer division. An extension to parallelism with asynchronous communication via channels is presented in Section 4. In Section 5 we give a top-down derivation of a distributed real-time control system in PVS, namely the chemical batch processing system (inspired by a description of this example in [1]). Concluding remarks can be found in Section 6.

2 A Mixed Formalism for Sequential Programs

In this section we consider only sequential real-time programs and define our mixed formalism in the PVS specification language. In general, a PVS specification consists of a number of theories. A theory can import other theories. In Section 2.1 we formulate the basic theory defining values and time constructs. Section 2.2 contains the main points of a theory for sequential real-time programs. Specifications and program refinement are defined in Section 2.3. A small example of semantic reasoning and the use of the PVS proof checker can be found in Section 2.4.

2.1 Values and Time

In this paper we consider a domain of values which equals the real numbers. In the PVS theory `rtcalc` below this is specified by defining the type `Value` to be equal to the built-in type `real`. As a time domain, represented by `Time`, we use the nonnegative reals. Further we define time intervals, using `co` to represent left-closed right-open intervals, etc. The types `setof[Time]` and `pred[Time]` are equivalent to the type `[Time -> bool]` denoting functions from `Time` to the built-in type `bool`.

The standard PVS operators `NOT, AND, OR, IMPLIES` on `bool` are overloaded in `rtcalc` and now also defined on predicates over `Time`. (The semicolon after the definition is needed to avoid ambiguity for the infix operators.) Finally we define when a time predicate holds *inside* or *during* an interval.

```
rtcalc : THEORY
BEGIN
Value    : TYPE = real
Time     : TYPE = { r : real | r >= 0 }
Interval : TYPE = setof[Time]

t        : VAR Time
v0 , v1  : VAR Value

cc ( v0 , v1 ) : Interval = { t | v0 <= t AND t <= v1 }
co ( v0 , v1 ) : Interval = { t | v0 <= t AND t < v1 }
oc ( v0 , v1 ) : Interval = { t | v0 < t AND t <= v1 }
oo ( v0 , v1 ) : Interval = { t | v0 < t AND t < v1 }

P , Q    : VAR pred[Time]

NOT ( P )           : pred[Time] = (LAMBDA t : NOT P(t)) ;
AND ( P , Q )       : pred[Time] = (LAMBDA t : P(t) AND Q(t)) ;
OR ( P , Q )        : pred[Time] = (LAMBDA t : P(t) OR Q(t)) ;
IMPLIES ( P , Q )   : pred[Time] = (LAMBDA t : P(t) IMPLIES Q(t)) ;

I        : VAR Interval

inside ( P , I ) : bool = (EXISTS t : I(t) AND P(t))
dur ( P , I )    : bool = (FORALL t : I(t) IMPLIES P(t))

END rtcalc
```

The PVS parser and typechecker can be applied to such a theory to check syntactic and semantic consistency.

2.2 Sequential Programs

Next we introduce sequential real-time programs in the theory `programs` which imports the theory `rtcalc`. The set of program variables `Vars` is introduced here as a parameter of the theory. The reason for this will be discussed in Section 2.4.

For the definition of programs in PVS there are several possibilities. For instance, on could define the syntactic structure of programs as an abstract datatype. PVS supports a powerful mechanism for abstract datatypes, including the generation of a function for inductive definitions on the datatype. This function can be used to define the semantics of programs by structural induction. After some experiments with this approach we found it simpler, and more flexible, to identify programs and their semantics. Hence in this paper a (real-time) program is simply a relation on states, i.e., a function from pairs of states to `bool` in PVS.

A state is a record with three fields: a `val` field which gives the values of variables, a `now` field which records the current time, and a `term` field which is used to indicate termination. For a state `s`, these fields are denoted by, respectively, `val(s)`, `now(s)`, and `term(s)`.

```
programs [ Vars : TYPE ] : THEORY
BEGIN
IMPORTING rtcalc

State : TYPE = [# val : [Vars->Value], now : Time, term : bool #]

program : TYPE = [ State , State -> bool ]
```

Henceforth we use the following variables:

```
v , v0 , v1 , v2      : VAR Value
t , t0 , t1 , t2      : VAR Time
s , s0 , s1 , s2      : VAR State
prog, prog1, prog2    : VAR program
b                     : VAR [State -> bool]
vvar                  : VAR Vars
exp                   : VAR [State -> Value]
```

Atomic actions are defined as a relation between initial state `s0` and final state `s1`. When defining programming constructs we will only specify the case that the statement starts in a terminated state, i.e., `term(s0)` holds. For the case that `term(s0)` does not hold we give a general axiom, labelled `nonterm_ax`, which specifies that a nonterminating state is not changed. (This is used later to obtain a convenient formulation of sequential composition.)

```
nonterm_ax : AXIOM NOT term(s0) IMPLIES (prog (s0,s1) IFF s0 = s1)
```

Free variables are implicitly universally quantified. E.g., axiom `nonterm_ax` is equivalent to `(FORALL prog, s0, s1 : NOT term(s0) IMPLIES ... )`.

Below we define a few programming constructs in PVS. The execution time of an assignment is represented by a constant `Ta`. Further the override expression `val(s0) WITH [(vvar) := exp(s0)]` denotes the function from `Vars` to `Value` which is the same as `val(s0)` except that the value of `vvar` is given by `exp(s0)`.

```
Ta       : Time

assign(vvar,exp) : program = (LAMBDA s0 , s1 : term(s0) IMPLIES
   term(s1) AND val(s1) = val(s0) WITH [(vvar) := exp(s0)] AND
   now(s1) = now(s0) + Ta)

delay(exp) : program = (LAMBDA s0 , s1 : term(s0) IMPLIES
   term(s1) AND val(s1) = val(s0) AND
   now(s1) = now(s0) + IF exp(s0) >= 0 THEN exp(s0) ELSE 0 ENDIF)

seq(prog1,prog2) : program = (LAMBDA s0 , s1 : term(s0) IMPLIES
   (EXISTS s : prog1(s0,s) AND prog2(s,s1)) )

ifthen(b,prog) : program = (LAMBDA s0 , s1 : term(s0) IMPLIES
   IF b(s0) THEN prog(s0,s1) ELSE s1 = s0 ENDIF )
```

In the semantics of a while statement we use a constant `Tw`, representing the time it takes to evaluate the boolean condition, and the built-in (polymorphic) type `sequence` of infinite sequences. The semantics of `while(b,prog)` is described using a sequence of states, representing executions of `prog` after evaluation of `b`. We distinguish three cases: termination after `k` iterations because `b` evaluates to false, nontermination after `k` iterations because `prog` does not terminate, and nontermination because `b` never evaluates to false and `prog` always terminates.

```
Tw       : Time

while(b,prog) : program = (LAMBDA s0 , s1 : term(s0) IMPLIES
   (EXISTS (ss :sequence[State]) : s0 = ss(0) AND
     ((EXISTS (k : nat) :
        (FORALL (j : nat) : j < k IMPLIES
                seq(delay(LAMBDA s : Tw), prog)(ss(j),ss(j+1)) AND
                b(ss(j)) AND term(ss(j)) AND
        ((term(ss(k)) AND NOT b(ss(k)) AND
                          delay(LAMBDA s : Tw)(ss(k),s1))
         OR (NOT term(ss(k)) AND s1 = ss(k)) ) ) )
     OR
     (NOT term(s1) AND (FORALL (j : nat) :
                seq(delay(LAMBDA s : Tw), prog)(ss(j),ss(j+1)) AND
                b(ss(j)) AND term(ss(j)) ) ) ) ) )
```

Given these definitions we can prove certain semantic properties. E.g., for sequential composition we can use axiom `nonterm_ax` to obtain the following lemma with label `seq_prop`:

```
seq_prop : LEMMA seq(prog1,prog2)(s0,s1) IFF
                 (EXISTS s : prog1(s0,s) AND prog2(s,s1))
```

How to prove properties in PVS will be explained in Section 2.4.

2.3 Specifications

To specify real-time systems we use assertions which are predicates over states. The logical connectives are also overloaded for state predicates, and we define a notion of validity. To support the mixed approach, a specification is also considered as a program, i.e., a relation on states. A specification is a pair (A,C) with the meaning that if the initial state satisfies assumption A then the final state should satisfy commitment C.

```
A , A1 , A2 , C   : VAR pred[State]

true              : pred[State] = (LAMBDA s : true)
false             : pred[State] = (LAMBDA s : false)
NOT (A)           : pred[State] = (LAMBDA s : NOT A(s)) ;
AND (A1, A2)      : pred[State] = (LAMBDA s : A1(s) AND A2(s)) ;
OR (A1, A2)       : pred[State] = (LAMBDA s : A1(s) OR A2(s)) ;
IMPLIES (A1, A2) : pred[State] = (LAMBDA s : A1(s) IMPLIES A2(s));
Valid (A)         : bool        = (FORALL s : A(s)) ;

spec ( A , C )    : program = (LAMBDA s0, s1 : A(s0) IMPLIES C(s1))
```

As usual one should be able to express that a program satisfies a specification, and in general, that one program refines another. For refinement we overload the infix operator => and prove a few simple properties.

```
=> : [ program , program -> bool ] =
       (LAMBDA prog1 , prog2 :
         (FORALL s0, s1 : prog1 (s0,s1) IMPLIES prog2 (s0,s1) )) ;

ref_refl  : THEOREM prog => prog

ref_trans : THEOREM (prog0 => prog2) IFF
            (EXISTS prog1 : (prog0 => prog1) AND (prog1 => prog2))
```

2.4 Example Semantic Reasoning

We give a simple example to show the notation in an example and the proof of a refinement relation using semantic reasoning.

```
rtex1 : THEORY
BEGIN
Vars : TYPE = {x,y}
```

```
IMPORTING programs [Vars]
s   : VAR State

A : pred[State] = (LAMBDA s : val(s)(x) = 1 AND val(s)(y) = 2 AND
                              now(s) = 5 AND term(s))

C : pred[State] = (LAMBDA s : val(s)(x) = 6 AND val(s)(y) = 2 AND
                              now(s) = 5 + Ta AND term(s))

expr : [State -> Value] = (LAMBDA s : val(s)(x) + val(s)(y) + 3)

cor1 : THEOREM assign(x,expr) => spec( A , C )
END rtex1
```

To prove `cor1`, one can use a PVS command which creates a new EMACS buffer (PVS uses EMACS as its interface), displays the formula, and asks the user for a command by the `Rule?` prompt.

```
cor1 :

  |-------
{1}  assign(x, expr) => spec(A, C)

Rule?
```

Now a proof command of PVS can be invoked. Typing `(expand "=>")` leads to

```
Rule? (expand "=>")
Expanding the definition of =>
this simplifies to:
cor1 :

  |-------
{1}    (FORALL (s0: State[Vars, Chan]), (s1: State[Vars, Chan]):
         assign(x, expr)(s0, s1) IMPLIES spec(A, C)(s0, s1))
```

By the `(skolem!)` command we can introduce Skolem constants `s0!1` and `s1!1` for `s0` and `s1` and then apply `(flatten)`:

```
cor1 :

  |-------
{1}    assign(x, expr)(s0!1, s1!1) IMPLIES spec(A, C)(s0!1, s1!1)

Rule? (flatten)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
cor1 :
```

```
{-1}    assign(x, expr)(s0!1, s1!1)
  |-------
{1}    spec(A, C)(s0!1, s1!1)
```

Expanding the definitions of `spec`, `assign`, `expr`, `A`, and `C`, and applying the command `(flatten)` this leads to

```
cor1 :

[-1]    term(s0!1)
        IMPLIES val(s1!1) =
             val(s0!1) WITH [x := val(s0!1)(x) + val(s0!1)(y) + 3]
          AND now(s1!1) = now(s0!1) + Ta AND term(s1!1)
{-2}    val(s0!1)(x) = 1
{-3}    val(s0!1)(y) = 2
{-4}    now(s0!1) = 5
{-5}    term(s0!1)
  |-------
{1}    val(s1!1)(x) = 6 AND val(s1!1)(y) = 2
        AND now(s1!1) = 5 + Ta AND term(s1!1)
```

Reasoning in PVS is based on the sequent calculus; the sequent above consists of antecedents numbered `-1` through `-5` and a succedent numbered `1`.

The current proof can now be finished by invoking the PVS decision procedures which, e.g., can automatically decide certain fragments of arithmetic. In this case, application of `(ground)` proves the succedent.

It is important to note that the proof of `val(s1!1)(y) = 2` requires that `x` and `y` are different variables, since the value of `x` is changed by the override expression in `-1`. Therefore we have defined `Vars` as an enumeration type {`x,y`}, since in PVS this implies that the identifiers `x` and `y` are distinct. Typechecking of the enumeration type generates the axiom `x /= y` which is used automatically by the decision procedures. (In fact, an enumeration type is a special case of the datatype mechanism in PVS.)

In a preliminary version of this work we have defined variables as an uninterpreted type and `x` and `y` as constants of that type, i.e.

```
Vars   : TYPE
x , y  : Vars
```

Then, however, one has to provide the axiom `x /= y` explicitly (and, moreover, this has to be done for each pair of variables).

Finally, note that the proof of `cor1` essentially expands the definitions, Skolemizes universally quantified variables, and invokes the decision procedures. This turns out to be a general approach for the verification of sequential programs without while constructs. Therefore we have defined in PVS a strategy, called `(seqprog)`, which performs these steps. In this strategy first the definitions of the current theory, the overloaded operators `NOT`, etc. for assertions, and the definitions of programming constructs are declared as automatic rewrite rules for the decision procedures. Next `=>` is expanded. Then Skolemization and the invocation of decision procedures is repeated until nothing changes. With strategy `(seqprog)` theorem `cor1` is proved automatically.

3 Proof Rules for Sequential Programs

Semantic reasoning, as done in the previous section, is not suitable for top-down program design where one would like to reason with the specifications of components without knowing their implementation. Therefore we derive proof rules for compound programming constructs using specifications of the components. The proof rules below are formulated as theorems in PVS, that is, they are proved by means of the semantic definitions.

In addition to the variables of the previous section, we use

```
A , A0 , A1 ,  B , C , C0 , C1   : VAR pred[State]
```

First we formulate a general consequence rule which allows strengthening of assumptions and weakening of commitments.

```
rules [ Vars : TYPE ] : THEORY
BEGIN
IMPORTING programs [Vars]

consequence : THEOREM Valid(A IMPLIES A0) AND Valid(C0 IMPLIES C)
                       IMPLIES
                      (spec( A0, C0 ) => spec( A, C ))
```

For sequential composition and choice we can derive theorems which reflect the classical proof rules of Hoare logic.

```
seq_comp_rule : THEOREM seq( spec( A, B ), spec( B, C ) )
                         => spec( A, C )

if_then_rule : THEOREM ifthen( b, spec( A AND b, C ) )
                        => spec( A, C OR (A AND NOT b) )
```

Clearly the rule for the while construct is more complicated, since in our framework also timing and progress properties can be expressed. First we define, for an assertion `I0 : VAR pred[State]`, a state predicate `infinite (I0)` which holds if `I0` allows arbitrary large values of `now`.

```
infinite (I0) : pred[State] = (LAMBDA s :
  (FORALL t1 : (EXISTS t2 : t2 > t1 AND I0 (s WITH [now := t2]))))
```

Then we formulate the while rule using a loop invariant `I`, an assertion `I0` which follows from `I` but should not restrict program variables or `term`, and assertion `Cterm` which holds if the while program terminates.

```
while_rule : THEOREM
(seq(delay(LAMBDA s : Tw), prog) => spec(I AND b AND term, I)) AND
(delay(LAMBDA s : Tw) => spec(I AND NOT b AND term, Cterm)) AND
(FORALL s0, s1 : I(s0) AND now(s0) = now(s1) IMPLIES I0(s1))
IMPLIES
while(b, prog) => spec( I, Cterm OR (I AND NOT term) OR
                            (infinite(I0) AND NOT term) )
```

To prove soundness of this while rule we need an axiom to express that programs never decrease time. Further we require that `Tw` is positive (to be able to prove progress properties) and postulate the axiom of Archimedes, since this is not part of the built-in properties of reals in PVS.

```
now_ax : AXIOM prog(s0,s1) AND term(s0) IMPLIES now(s1) >= now(s0)

Tw_pos : AXIOM Tw > 0

archim : AXIOM (FORALL (epsilon : Time) : epsilon > 0 IMPLIES
                (FORALL t : (EXISTS (k : nat) : k * epsilon > t)))
```

Next we give a few examples of monotonicity properties which are needed to formalize top-down design.

```
mono_seq : THEOREM (prog3 => prog1) AND (prog4 => prog2)
                    IMPLIES
                   (seq(prog3,prog4) => seq(prog1,prog2))

mono_while : THEOREM (prog => prog0)
                      IMPLIES
                     (while(b,prog) => while(b,prog0))
```

Although the framework given above is intended for the verification of timing properties, it includes a mixed formalism for the partial correctness of non-real-time programs. We formulate a theory `Hoare_logic` which can be applied if assertions do not refer to timing or termination, as characterized by predicate `nonrt`.

```
Hoare_logic [ Vars : TYPE ] : THEORY
BEGIN
IMPORTING rules [Vars]

p     : VAR pred[State]

nonrt ( p ) : bool = (FORALL s0 , s1 :
                        val(s0) = val(s1) AND p(s0) IMPLIES p(s1))
```

The definition of `nonrt(p)` expresses that assertion `p` only depends on the `val` field of a state, that is, it does not restrict the `now` and `term` fields. Then we can prove the classical while rule.

```
while_nonrt : THEOREM nonrt(p) IMPLIES
              while( b, spec( p AND b AND term, term IMPLIES p ) )
              =>
              spec( p,  term IMPLIES p AND NOT b )
```

3.1 Example Integer Division

As an illustration of top-down design of non-real-time programs in this mixed approach we derive a simple program for integer division. The aim is to design a program which computes, for given `x` and `y`, values for the variables `z` and `r` such that $x = z \times y + r$ with $0 \leq r < y$.

```
ex_div : THEORY
BEGIN
Variables : TYPE = {x,y,z,r}
IMPORTING Hoare_logic[Variables]

p : pred[State] = (LAMBDA s: val(s)(x) >= 0 AND term(s))

q : pred[State] = (LAMBDA s:
              val(s)(x) = val(s)(z) * val(s)(y) + val(s)(r) AND
              0 <= val(s)(r) AND val(s)(r) < val(s)(y) )
```

Then partial correctness is specified by `spec(p, term IMPLIES q)`. To implement this by means of a while loop, we transform postcondition `q` into a loop invariant `inv` and take the negation of part of the postcondition as the boolean condition of a while construct. Here we define

```
inv : pred[State] = (LAMBDA s:
                val(s)(x) = val(s)(z) * val(s)(y) + val(s)(r) AND
                0 <= val(s)(r) )

b   : pred[State] = (LAMBDA s : val(s)(r) >= val(s)(y) )
```

Then the desired program can be split up into a part which realizes `inv` and a second part which leads to `q` in case of termination. We show, using rule `while_nonrt`, that the second part can be implemented by a while construct.

```
cor_top : LEMMA
          seq( spec( p, inv ), spec( inv, term IMPLIES q ) )
          => spec( p, term IMPLIES q )

cor_whi : LEMMA
          while( b, spec( inv AND b AND term, term IMPLIES inv ) )
          => spec( inv, term IMPLIES q )
```

Next the initial part and the body of the while construct are implemented by assignments.

```
init : program = seq( assign( z, LAMBDA s : 0),
                      assign( r, LAMBDA s : val(s)(x)) )

cor_init : LEMMA init => spec( p, inv )

body : program = seq( assign( r, LAMBDA s : val(s)(r)-val(s)(y)),
                      assign( z, LAMBDA s : val(s)(z)+1) )

cor_body : LEMMA
           body => spec( inv AND b AND term, term IMPLIES inv )
```

The lemmas `cor_init` and `cor_body` can be proved automatically by strategy `(seqprog)` mentioned at the end of Section 2.4.
Using monotonicity properties and transitivity of `=>` the lemmas above lead to

```
cor_part : THEOREM seq( init, while( b, body ) )
                   => spec( p, term IMPLIES q )
```

The proof of this theorem can also be automated by defining a strategy which, among others, parses the goal to be able to apply the right monotonicity rule.

It is easy to prove that the resulting program does not change `x` and `y`.

```
x0 , y0 : Value

pfreeze : pred[State] = (LAMBDA s:
                    val(s)(x) = x0 AND val(s)(y) = y0 AND term(s))

qfreeze : pred[State] = (LAMBDA s:
                    val(s)(x) = x0 AND val(s)(y) = y0)

cor_freeze : THEOREM seq( init, while( b, body ) )
                     => spec( pfreeze, term IMPLIES qfreeze )
```

Next we show that we can also prove timing properties of this program. For instance, that termination implies a certain termination time (termination itself is proved later).

```
prt : pred[State] = (LAMBDA s: now(s) = 0 AND term(s))

qrt : pred[State] = (LAMBDA s:
                 now(s) = 2 * Ta + val(s)(z) * (Tw + 2 * Ta) + Tw)

cor_rt : THEOREM seq( init, while( b, body ) )
                 => spec( prt, term IMPLIES qrt )
```

This theorem has been proved with `while_rule` using the following invariant.

```
invrt : pred[State] = (LAMBDA s: term(s) AND
                      now(s) = 2 * Ta + val(s)(z) * (Tw + 2 * Ta))
```

The next aim is to show termination. Essentially this is done by showing that `now` is bounded. For simplicity, we assume that `y` is positive. Define

```
pxy : pred[State] = (LAMBDA s: x0 >= 0 AND y0 > 0 AND term(s))

pterm : pred[State] = p AND prt AND pfreeze AND pxy
```

To prove nontermination, i.e., `spec(pterm, term)`, let

```
invxyz : pred[State] =  (LAMBDA s:
                           x0 >= 0 AND y0 > 0 AND val(s)(z) >= 0)

invterm : pred[State] = inv AND invrt AND qfreeze AND invxyz
```

Then we can show that `invterm` is an invariant of the while construct, and that `invterm` implies `I0` which is defined by

```
I0 : pred[State] = (LAMBDA s: x0 >= 0 AND y0 > 0 AND
                      now(s) <= 2 * Ta + (x0/y0) * (Tw + 2 * Ta) )
```

It is not difficult to prove

```
I0_lem : LEMMA Valid( infinite(I0) IMPLIES false )
```

and then `while_rule` leads to

```
cor_term : THEOREM seq( init, while( b, body ) )
                   => spec( pterm, term )
```

Combining the results above we obtain

```
A : pred[State] = (LAMBDA s: val(s)(x) = x0 AND val(s)(y) = y0 AND
                    x0 >= 0 AND y0 > 0 AND term(s) AND now(s) = 0)

C : pred[State] = (LAMBDA s: val(s)(x) = x0 AND val(s)(y) = y0 AND
                x0 = val(s)(z) * y0 + val(s)(r) AND
                0 <= val(s)(r) AND val(s)(r) < y0 AND term(s) AND
                now(s) = (val(s)(z) + 1) * (Tw + 2 * Ta))

cor_tot : THEOREM seq( init, while( b, body ) )
                  => spec( A , C )
END ex_div
```

4 Parallel Programs

In this section we extend our approach to parallel programs which communicate via message passing along unidirectional channels. Communication is asynchronous, that is, a sender does not wait for synchronization but sends the message immediately. A receiver waits until a message is available. We assume that there is no buffering of messages; a message gets lost if there is no receiver.

Similar to the treatment of program variables, it is convenient to define channels as an enumeration type in examples. Hence the theory `programs` is extended with a parameter for the set of channels. First we define primitives to describe asynchronous communication, that is, to express when a process starts sending a value, when it is waiting to receive, and when it starts receiving a value.

```
programs [ Vars : TYPE , Chan : TYPE ] : THEORY

sendv     : [ Chan,Value -> pred[Time] ]
waitrec   : [ Chan       -> pred[Time] ]
recv      : [ Chan,Value -> pred[Time] ]
```

It is often convenient to abstract from the values communicated.

```
ch       : VAR Chan
send(ch) : pred[Time] = (LAMBDA t: (EXISTS v : sendv(ch,v)(t)))
rec(ch)  : pred[Time] = (LAMBDA t: (EXISTS v : recv(ch,v)(t)))
```

Next we define input and output statements, again by identifying them with their semantics. Note that an input statement need not terminate because it might have to wait forever.

```
Tc      : Time
Tc_pos  : AXIOM Tc > 0

output ( ch, vvar ) : program = (LAMBDA s0, s1 : term(s0) IMPLIES
 term(s1) AND val(s1) = val(s0) AND
 sendv(ch,val(s0)(vvar))(now(s0)) AND now(s1) = now(s0) + Tc )

input ( ch, vvar ) : program = (LAMBDA s0, s1 : term(s0) IMPLIES
 (NOT term(s1) AND
  (FORALL t : t >= now(s0) IMPLIES waitrec(ch)(t)) )
 OR
 (term(s1) AND now(s1) - Tc >= now(s0) AND
  dur( waitrec(ch), co( now(s0), now(s1) - Tc ) ) AND
  (EXISTS v : val(s1) = val(s0) WITH [ (vvar) := v ] AND
              recv(ch,v)(now(s1) - Tc)) ))
```

Further we have to extend the meaning of constructs to be able to show that nothing happens on a channel during certain periods of time. A few examples:

```
noio ( t0 , t1 ) : bool = (FORALL ch :
           dur( NOT rec(ch) AND NOT send(ch) AND NOT waitrec(ch) ,
                co( t0 , t1 )) )

chan_inv_assign : AXIOM assign(vvar,exp)(s0,s1) AND term(s0)
```

```
                        IMPLIES noio ( now(s0) , now(s1) )

chan_inv_delay : AXIOM delay(exp)(s0,s1) AND term(s0)
                       IMPLIES noio ( now(s0) , now(s1) )
```

We do not give the semantics of parallel composition here, but directly formulate the rule for parallel composition as an axiom. Also, for simplicity, we have omitted the syntactic constraints which require that the assertions of one process do not refer to observables of the other process. Additionally, assume that `now` and `term` do not occur in the commitments. We refer to [3] for more details and a soundness proof of the parallel composition rule. Here we concentrate on the use of this rule during top-down program design of distributed systems.

```
rules [ Vars : TYPE , Chan : TYPE ] : THEORY
BEGIN
IMPORTING programs [Vars,Chan]

par_comp : AXIOM  par( spec(A1,C1), spec(A2,C2) )
                  => spec( A1 AND A2, C1 AND C2 )
```

Moreover there is a monotonicity axiom for parallel composition.

By `par_comp` we can combine assertions about input and output actions on a particular channel, and for further reasoning we need axioms to relate the communication primitives. In the theory `asyn` we axiomatize the properties of asynchronous communication.

```
asyn [ Vars : TYPE , Chan : TYPE ] : THEORY
BEGIN
IMPORTING rules [Vars,Chan]

val_id : AXIOM sendv(ch,v1)(t) AND sendv(ch,v2)(t) IMPLIES v1 = v2

min_wait : AXIOM NOT ( send(ch)(t) AND waitrec(ch)(t) )

rec_send : AXIOM recv(ch,v)(t) IMPLIES sendv(ch,v)(t)
```

Next we define a few useful abbreviations.

```
awaitrec (ch) : pred[Time] = (LAMBDA t :
           (FORALL t1 : t1 >= t IMPLIES waitrec(ch)(t1)) OR
           (EXISTS t1 : t1 >= t AND dur(waitrec(ch), co(t,t1)) AND
                                     rec(ch)(t1)))

Start , Period , T , T1 , T2   : VAR Time

maxsend (ch,Start,Period) : pred[Time] = (LAMBDA t :
  send(ch)(t) IMPLIES
              t >= Start AND dur( NOT(send(ch)), oo(t-Period,t) ))
```

```
minawait (ch,Start,Period) : pred[Time] = (LAMBDA t :
  t >= Start IMPLIES inside( awaitrec(ch), oc(t-Period,t) ) )
```

The next lemma expresses that under certain conditions no message gets lost.

```
send_rec : LEMMA (FORALL t : maxsend(ch,Start,Period)(t)) AND
                 (FORALL t : minawait(ch,Start,Period)(t))
                  IMPLIES
                 (FORALL t : send(ch)(t) IFF rec(ch)(t))
```

Finally we give a few abbreviations and a lemma to express that a message is received at least once in a certain period of time.

```
sendperiod(ch,T)(t) : bool = inside( send(ch), co(t,t+T) )

waitperiod(ch,T)(t) : bool = inside( awaitrec(ch), co(t,t+T) )

commperiod(ch,T)(t) : bool = inside( rec(ch), co(t,t+T) )

comm_per_lem : LEMMA (FORALL t : sendperiod(ch,T1)(t)) AND
                     (FORALL t : waitperiod(ch,T2)(t))
                      IMPLIES
                     (FORALL t : commperiod(ch,T1+T2)(t))
END asyn
```

5 Example Chemical Batch Processing

To illustrate top-down design of distributed real-time systems in our framework, we consider a chemical batch processing example which is inspired by a description in [1]. It consists of a batch processing plant which has a reaction vessel filled with chemicals. Heating two chemicals produces a third chemical which is hazardous and might lead to an explosion. We use a time predicate `expl` to denote that an explosion occurs in the vessel at a certain point of time.

```
chem : THEORY
BEGIN
Vars : TYPE = { x }
Chan : TYPE = { thermchan , actchan }
IMPORTING asyn [Vars,Chan]

expl : pred[Time]
```

The top-level specification of the chemical batch processing system requires that there should be no explosion, expressed by `spec(A, CTL)` with

```
A : pred[State] = (LAMBDA s : now(s) = 0 AND term(s))

CTL : pred[State] = (LAMBDA s : (FORALL t : NOT expl(t)))
```

To implement a control system which establishes this property, we first specify a physical property of the chemicals in the vessel. Suppose some chemical analysis yields that there will be no explosion if the temperature is below a certain value, say `ExpTemp`, or if the vessel is empty.

```
temp       : [ Time -> Value ]    % temperature in vessel
empty      : pred[Time]           % empty vessel
ExpTemp    : Value                % explosion temperature

CV : pred[State] = (LAMBDA s : (FORALL t :
             temp(t) <= ExpTemp OR empty(t) IMPLIES NOT expl(t) ))
```

Given this property, there are several possible strategies for a control system when it detects that the temperature is too high. For instance it might cool the chemicals while they are in the vessel. Here we follow [1] and decide to empty the contents into a cooled vat. This strategy is specified by the commitment

```
CS : pred[State] = (LAMBDA s :
                 (FORALL t : temp(t) > ExpTemp IMPLIES empty(t) ))
```

The correctness of this design step is formulated by the following theorem, which is proved by axiom `par_comp` and lemma `cor_CTL`.

```
cor_CTL : LEMMA Valid( CV AND CS IMPLIES CTL )

cor_TL  : THEOREM par( spec(A,CV), spec(A,CS) ) => spec(A,CTL)
```

Suppose we have a thermometer which measures the temperature and sends the measured values along channel `thermchan`. We assume that the value of the thermometer does not deviate more than `ThermDev` from the real temperature. Here we only need an upper bound. The thermometer will send values at least once every `DelTherm` time units. Further there are two assumptions about the maximal change of the temperature, using a positive parameter `MaxRise`, and the initial temperature at time `0`, using a safety temperature `SafeTemp`.

```
ThermDev       : Value     % deviation of thermometer
DelTherm       : Time      % delay of thermometer
MaxRise        : Value     % max rise of temperature per second
SafeTemp       : Value     % safety temperature

CSEN1 : pred[State] = (LAMBDA s : (FORALL t, v :
        sendv(thermchan,v)(t) IMPLIES v > temp(t) - ThermDev ))

CSEN2 : pred[State] = (LAMBDA s : (FORALL t :
                              sendperiod(thermchan,DelTherm)(t) ))

CSEN3 : pred[State] = (LAMBDA s : (FORALL t0, t1 : t0 < t1 IMPLIES
                    temp(t1) - temp (t0) <  MaxRise * (t1 - t0) ))
```

```
CSEN4 : pred[State] = (LAMBDA s : temp(0) <= SafeTemp + ThermDev )

CSEN : pred[State] = CSEN1 AND CSEN2 AND CSEN3 AND CSEN4
```

To obtain specification `spec(A,CS)`, we design in parallel with the thermometer a flow control component which is ready to receive input from the sensor along `thermchan` at least once every `DelReadTherm` time units. Further we specify that when an unsafe temperature is detected, i.e. above safety temperature `SafeTemp`, the vessel is emptied in at most `DelEmpty` time units.

```
DelReadTherm , DelEmpty   : Time

CFC1 : pred[State] = (LAMBDA s : (FORALL t :
                          waitperiod(thermchan,DelReadTherm)(t) ))

EmptyVessel : pred[Time] = (LAMBDA t0 :
                       (FORALL t1 : t1 >= t0 IMPLIES empty(t1) ))

CFC2 : pred[State] = (LAMBDA s : (FORALL t, v :
   recv(thermchan,v)(t) AND v > SafeTemp IMPLIES
               inside ( EmptyVessel , cc(t,t+DelEmpty) ) ))

CFC : pred[State] = CFC1 AND CFC2
```

For the correctness of this step, first define

```
COMTHERM : pred[State] = (LAMBDA s :
     (FORALL t : commperiod(thermchan,DelTherm+DelReadTherm)(t) ))
```

Then `CSEN2` and `CFC1` lead by lemma `comm_per_lem` of theory `asyn` to obtain:

```
comm_thermchan : LEMMA Valid( CSEN2 AND CFC1 IMPLIES COMTHERM )
```

Further we need a relation between timing parameters and an axiom.

```
TDbound : bool = ExpTemp - (SafeTemp + ThermDev) >=
                 (DelTherm + DelReadTherm + DelEmpty) * MaxRise

MR_pos : AXIOM MaxRise > 0
```

Then we can prove

```
corCS: LEMMA TDbound AND DelEmpty > 0 IMPLIES
             Valid( CSEN AND CFC IMPLIES CS )

corS : THEOREM TDbound AND DelEmpty > 0 IMPLIES
               par( spec(A,CSEN) , spec(A,CFC) )
               => spec(A,CS)
```

To implement `spec(A,CFC)` we use an actuator to empty the vessel. Suppose the actuator can be activated by sending a message along channel `actchan`. We assume that the actuator is ready to receive input periodically, using parameters `InitialPeriod` and `RepPeriod`. Further the actuator will respond to a signal along `actchan` by emptying the vessel in at most `DelAct` time units.

```
InitialPeriod , RepPeriod , DelAct   : Time

CA1 : pred[State] = (LAMBDA s :
      (FORALL t : minawait(actchan,InitialPeriod,RepPeriod)(t) ))

CA2 : pred[State] = (LAMBDA s :
      (FORALL t : rec(actchan)(t) IMPLIES
                  inside ( EmptyVessel , cc(t,t+DelAct) ) ))

CA : pred[State] = CA1 AND CA2
```

In parallel with this actuator we design a control component which sends signals to the actuator along `actchan`. To guarantee that no message gets lost, we specify, in view of `CA1`, the maximal frequency with which it will send messages along `actchan`. The main task of the control component is to send a signal along `actchan`, in at most `DelContr` time units, if it receives a value via `thermchan` which is greater than `SafeTemp`.

```
DelContr      : Time

CC1 : pred[State] = (LAMBDA s :
        (FORALL t : maxsend(actchan,InitialPeriod,RepPeriod)(t) ))

CC2 : pred[State] = (LAMBDA s : (FORALL t, v :
           recv(thermchan,v)(t) AND SafeTemp < v IMPLIES
                    inside ( send(actchan) , cc(t,t+DelContr) ) ))

CC : pred[State] = CC1 AND CC2 AND CFC1
```

Further we need the following relation between timing parameters.

```
Del_bound : AXIOM DelEmpty >= DelAct + DelContr
```

Note that `CA1` and `CC1` imply, by lemma `send_rec` of `asyn`, that no message along `actchan` gets lost. This leads to the correctness of this design step.

```
corCC: LEMMA Valid( CA AND CC IMPLIES CFC )

corC : THEOREM par( spec(A,CA) , spec(A,CC) )
               => spec(A,CFC)
```

It remains to design a program satisfying `spec(A,CC)`. Here we directly give a program, called `controlprog`, and prove its correctness.

```
high : pred[State] = (LAMBDA s : val(s)(x) > SafeTemp)

body : program = seq( input(thermchan,x),
                      ifthen(high, output(actchan,x)) )

controlprog :  program = while( true , body )
```

To prove that it meets the required specification, we define an invariant `Inv` using `ICC1`, `ICC2`, `ICFC1`, and `ICC` defined as follows.

```
ICC1 : pred[State] = (LAMBDA s :
   (FORALL t :  t <= now(s) - Tc IMPLIES
                maxsend(actchan,InitialPeriod,RepPeriod)(t) ) AND
                dur( NOT send(actchan) , oo(now(s)-Tc,now(s))  ))

ICC2 : pred[State] = (LAMBDA s :
   (FORALL t, val : t < now(s) IMPLIES
           recv(thermchan,v)(t) AND SafeTemp < v IMPLIES
                    inside ( send(actchan) , cc(t,t+DelContr) ) ))

ICFC1 : pred[State] = (LAMBDA s :
   (FORALL t : t <= now(s) - 2 * Tc IMPLIES
                          waitperiod(thermchan,DelReadTherm)(t) ))

ICC : pred[State] = ICC1 AND ICC2 AND ICFC1

Inv : pred[State] = (ICC AND term) OR (CC AND NOT term)
```

For the correctness proof we define the following values for the time parameters.

```
IP_bound  : AXIOM InitialPeriod = Tw + Tc

RP_bound  : AXIOM RepPeriod = Tw + 2 * Tc

DC_bound  : AXIOM DelContr = Tc

DRT_bound : AXIOM DelReadTherm = Tw + 2 * Tc
```

Then we can derive

```
cor_control : THEOREM controlprog => spec(A,CC)
```

which leads to the final conclusion:

```
cor_total : THEOREM SafeTemp <=
 ExpTemp - ThermDev - (DelTherm + 3 * Tc + Tw + DelAct) * MaxRise
 IMPLIES par( spec(A,CV) ,
         par( spec(A,CSEN) ,
         par( spec(A,CA) ,
              controlprog  ) ) ) => spec(A,CTL)
```

Hence we have obtained a system which implements the top-level specification, assuming specifications of the physical properties of the vessel, the thermometer, and the actuator, and provided `SafeTemp` is sufficiently smaller than the explosion temperature `ExpTemp` in order to cope with delays in the thermometer, the program, and the actuator, and with the maximal rise of the temperature.

6 Conclusion

We have presented a mixed formalism for the correct construction of distributed real-time systems. By defining the theory in PVS, proofs can be checked mechanically and simple details are proved automatically using the PVS decision procedures (and errors are found in apparently trivial details). This improves the speed of the design and the verification and allows the user to concentrate on the essential structure of proofs.

The possibility to build hierarchies of parameterized theories turns out to be very useful. In future work we intend to extend the framework with more theories for parallel programs, e.g. dealing with various communication mechanisms, and to add theories for general reasoning about real-time programs such as a calculus for time intervals. Using the powerful higher-order specification language of PVS it is easy to formulate general patterns and schemas.

Since we have identified programs and their semantics, we can easily define abstract statements which can be refined during later stages of the design process into concrete programming constructs. For instance, for an output statement we can abstract from the value transmitted and define

```
output(ch) : program = (LAMBDA s0, s1 : term(s0) IMPLIES
         term(s1) AND val(s1) = val(s0) AND send(ch)(now(s0)) AND
         now(s1) = now(s0) + Tc )
```

Then `output(ch, vvar) => output(ch)`. To give another example, we can define a statement which terminates between certain bounds:

```
bounds(t1,t2) : program = (LAMBDA s0 , s1 : term(s0) IMPLIES
 term(s1) AND now(s0) + t1 <= now(s1) AND now(s1) <= now(s0) + t2)
```

This makes it, for instance, possible to express general properties of a program of the form

```
while(true, seq(input(inch), seq( bounds(t1,t2), output(outch) )))
```

which represents a control loop that receives input, performs some computation and then produces output.

Another topic of future research is the development of a nice user interface which allows the use of the conventional notations for assertions and programs (hiding, e.g., the explicit references to the state in assertions). This is strongly related to the work presented in [10] where the PVS tool has been adapted to obtain a proof assistant for the Duration Calculus.

Acknowledgement

Many thanks go to Sreeranga Rajan for interesting discussions and valuable support on the use of PVS, expecially concerning the formulation of proof strategies.

References

1. T. Anderson, R. de Lemos, J.S. Fitzgerald, and A. Saeed. On formal support for industrial-scale requirements analysis. In *Workshop on Theory of Hybrid Systems*, pages 426–451. LNCS 736, 1993.
2. C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, 1969.
3. J. Hooman. *Specification and Compositional Verification of Real-Time Systems.* LNCS 558, Springer-Verlag, 1991.
4. J. Hooman. A compositional approach to the design of hybrid systems. In *Workshop on Theory of Hybrid Systems*, pages 121–148. LNCS 736, 1993.
5. J. Hooman. Compositional verification of a distributed real-time arbitration protocol. *Real-Time Systems*, 6:173–205, 1994.
6. J. Hooman. Extending Hoare logic to real-time. *Formal Aspects of Computing*, To appear, 1994.
7. C. Morgan. *Programming from Specifications.* Prentice Hall, 1990.
8. E. R. Olderog. Process theory: semantics, specification and verification. In *ESPRIT/LPC Advanced School on Current Trends in Concurrency*, pages 509–519. LNCS 194, Springer-Verlag, 1985.
9. S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In *11th Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, 1992.
10. J.U. Skakkebæk and N. Shankar. Towards a duration calculus proof assistant in PVS. In *Formal Techniques in Real-Time and Fault Tolerant Systems.* LNCS, This Volume, Springer-Verlag, 1994.