Developing Proof Rules for Distributed Real-Time Systems with PVS

Jozef Hooman

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

Abstract. In previous work, the verification system PVS has been used to support an assertional method for the specification and verification of distributed real-time systems. Essential part of the method is a compositional rule for parallel composition. In this paper we focus on the formalization of parallel composition in PVS. Two, equivalent, versions of the semantics of parallel composition are formulated in the specification language of PVS. Based on this semantics, several proof rules are shown to be sound, using the interactive proof checker of PVS. We indicate how the general framework can be instantiated for a particular class of applications by giving an axiomatization of asynchronous communication.

1 Introduction

Based on Hoare triples (precondition, program, postcondition), a formal framework for the specification and verification of distributed real-time systems has been developed [Hoo91]. This framework has been applied to a number of case studies, such as a water level monitoring system [Hoo93] and a distributed real-time arbitration protocol [Hoo94a].

Although these examples have been verified manually, it became clear that for more complex applications some form of mechanical support is indispensable. Therefore we have experimented with the verification system PVS[1] (Prototype Verification System) [ORS92, ORSvH95]. The PVS specification language is a higher-order typed logic, with many built-in types including booleans, integers, reals, sets, lists, and records. Further the tool contains a proof checker which makes it possible to construct proofs interactively and to rerun proofs automatically after small changes.

The main ingredients of our formal framework have been formulated in the language of PVS. To be able to represent design steps formally, specifications and programming constructs can be mixed freely, similar to the untimed framework of [Old85, Zwi89] and more recent work on timed systems [SO98]. This allows, e.g., the parallel composition of specifications. By means of refinement steps, justified by proof rules, specifications can be transformed into complete

[1] PVS is free available, see `http://www.csl.sri.com/sri-csl-pvs.html`

programs. The resulting PVS framework (see [Hoo94b]) has been applied to several examples, including the RPC-Memory specification problem [Hoo96], part of the ACCESS.bus protocol [Hoo95], a steam boiler control system [VH96], and a membership protocol for a dynamic network with node failures and local clocks [Hoo97][2].

As a start, these examples were verified using a simplified version of the formal framework. The semantics of sequential constructs has been formulated and proof rules (e.g. for iteration) have been proved as a theorem in PVS. The rule for parallel composition, however, was formulated as an axiom and the constraints on assertions were omitted and only checked manually. Since the case studies indicate that PVS provides convenient tool support, especially during the early stages of system design when dealing with parallel processes, it is important to improve the formulation of parallel composition in PVS.

This leads to the main goal of this paper; the design of a sound proof rule for the parallel composition of real-time processes. The rule should be proved as a theorem in PVS, based on a formal semantics. To support the idea of top-down design, the framework should allow mixed terms as mentioned above, that is, we should consider the parallel composition of real-time specifications. Further the intention is to give a general, abstract, framework, which can be instantiated for different examples with, e.g., different ways of communication between components.

To achieve this goal, we proceed as follows. Section 2 contains a number of primitives to express real-time specifications. Then, in section 3, the semantics of parallel composition is defined. To increase the confidence in the semantics, alternative definitions are presented and shown to be equivalent. Next, in section 4, several versions of the parallel composition rule are proved as a theorem in PVS. To show how this general framework can be used for a particular class of applications, we present in section 5 a few possibilities to formalize asynchronous communication. Throughout the paper, small examples will be given to illustrate the framework. Concluding remarks can be found in section 6.

2 Specification

We formulate a framework for the description of real-time components in the PVS specification language. PVS specifications can be structured into a hierarchy of parameterized theories. We start with a theory `RTPrim` which defines type `Time` using the built-in type `real`. Intervals are defined as sets of time points, where `setof[Time]` is equal to `[Time -> bool]`, i.e. a function from the time domain into the booleans. Intervals can be closed (`cc`), left-closed and right-open (`co`), etc. Finally we define when a predicate over time holds *inside* or *during* an interval. Note that `pred[Time]` is just another notation for `[Time -> bool]`.

```
RTPrim : THEORY
BEGIN
```

[2] Recent work is available on `http://www.win.tue.nl/~hooman`

```
Time       : TYPE = real

t, t0, t1 : VAR Time

Interval : TYPE = setof[Time]

cc(t0,t1) : Interval = { t | t0 <= t AND t <= t1 }
co(t0,t1) : Interval = { t | t0 <= t AND t < t1 }
oc(t0,t1) : Interval = { t | t0 < t AND t <= t1 }
oo(t0,t1) : Interval = { t | t0 < t AND t < t1 }

P, Q        : VAR pred[Time]
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 RTPrim
```

This theory is imported in theory `Specs` which contains the basic definitions for real-time components and specifications. To obtain a theory which can be applied in many different situations, it is parameterized by the types of the values, the variables and the events.

In previous work [Hoo94b] the semantics of sequential real-time programs has been formulated in terms of an initial state, which describes the situation at the start of the execution, and a final state for the state of affairs at termination. A `State` is represented by a record with three fields:

- `val` describes the values of the variables,
- `now` describes the starting or termination time; in the initial state it describes the starting time and in the final state it describes the termination time,
- `term` indicates termination of a component.

```
Specs [Values, Vars, Events : TYPE] : THEORY
BEGIN
IMPORTING RTPrim
State : TYPE =
            [# val : [Vars -> Values], now : Time, term : bool #]
```

The basic idea is that variables are local, i.e. not shared by parallel components, and used to transfer information sequentially. Parallel components communicate by means of events which can be used to synchronize, transfer messages, etc., depending on the communication mechanism. This external interface is represented by an observation function which assigns to each point of time the set of events that occur at that time. A component is described by a record with three fields:

- **alpha** describes the alphabet of the component, that is, the set of events that might occur during its execution,
- **vari** represents the set of local variables that might be changed by the component,
- **sem** describes the semantics of the component, represented by a relation between an initial state, a final state, and an observation function.

```
ObsFuncts : TYPE = [Time -> setof[Events]]
CompInfo  : TYPE = [# alpha : setof[Events],
                      vari  : setof[Vars],
                      sem   : [State, State, ObsFuncts -> bool]
                    #]
```

Components are characterized by a record of type `CompInfo` satisfying a few constraints. To express these constraints, first a number of variables are declared and some of the pre-defined operations for sets (`intersection`, `union`, `subset?`) are overloaded to observation functions.

```
s, s0, s1, s2       : VAR State
t, t0, t1, t2       : VAR Time
o, o1, o2           : VAR ObsFuncts
ci                  : VAR CompInfo
E                   : VAR Events
V                   : VAR Vars
Eset, Eset1, Eset2 : VAR setof[Events]
Vset, Vset1, Vset2 : VAR setof[Vars]

intersection(o,Eset) : ObsFuncts = LAMBDA t :
                                     intersection(o(t),Eset)
union(o1,o2)         : ObsFuncts = LAMBDA t : union(o1(t),o2(t))
subset?(o,Eset)      : bool = FORALL t : subset?(o(t),Eset)
```

Henceforth declarations of variables such as `s, s0`, etc., will not be repeated.

Next we express that a component can only generate events of its own alphabet and variables outside its own interface are not changed in case of termination.

```
CompInterface(Eset,Vset,s0,s1,o) : bool =
 subset?(o,Eset) AND
 (term(s0) AND term(s1) IMPLIES
  (FORALL V : NOT member(V,Vset) IMPLIES val(s0)(V) = val(s1)(V)))
```

The final state should be non-terminating if the initial state is non-terminating (which means that a sequentially preceding component did not terminate).

```
NonTerm(s0,s1)      : bool = NOT term(s0) IMPLIES NOT term(s1)
```

Further, in case of termination, the termination time should be greater or equal than the starting time.

```
TimeAdvance(s0,s1) : bool = term(s1) IMPLIES now(s1) >= now(s0)

CompProp(ci) : bool = FORALL s0, s1, o :
   sem(ci)(s0,s1,o) IMPLIES
               CompInterface(alpha(ci),vari(ci),s0,s1,o) AND
               NonTerm(s0,s1) AND TimeAdvance(s0,s1)

Components : TYPE = {ci | CompProp(ci)}
```

Next we define when one components implements another and prove that this refinement relation is reflexive and transitive.

```
comp, comp0, comp1, comp2, comp3 : VAR Components

=> (comp1, comp2) : bool =
              subset?(alpha(comp2),alpha(comp1)) AND
              subset?(vari(comp2),vari(comp1)) AND
              (FORALL s0, s1, o :
                sem(comp1)(s0,s1,o) IMPLIES sem(comp2)(s0,s1,o) )

RefRefl  : THEOREM comp => comp
RefTrans : THEOREM (comp0 => comp2) IFF
           (EXISTS comp1 : (comp0 => comp1) AND (comp1 => comp2))
```

All theorems and lemmas presented here have been proved by means of the proof checker of PVS. This proof checker is interactive, that is, the user tries to simplify the proof obligation by giving proof commands (e.g., to remove quantification, apply propositional and arithmetical simplification, use induction or case distinction).

Components can be represented by assertional specifications. Assertions are predicates over pairs consisting of a state and an observation function. The boolean connectives, `NOT`, `AND`, etc., are overloaded for assertions (showing a few cases only), and validity of assertions is defined.

```
Assertion : TYPE = pred[[State,ObsFuncts]]
A, A0, A1, A2, C, C0, C1, C2 : VAR Assertion

true    : Assertion = LAMBDA s, o : true
term    : Assertion = LAMBDA s, o : term(s)
NOT     : [Assertion -> Assertion] =
           LAMBDA A :  (LAMBDA s, o : NOT A(s,o) );
AND     : [Assertion, Assertion -> Assertion] =
           LAMBDA A1, A2 : (LAMBDA s, o : A1(s,o) AND A2(s,o) );
Valid(A) : bool = FORALL s, o : A(s,o)
```

Specifications consist of a set of observable events, a set of local variables and two assertions, an assumption `A` and a commitment `C`, which play a similar role as the well-known pre- and postconditions [Hoa69].

```
spec(Eset, Vset, A, C) : Components =
      (# alpha := Eset,
         vari  := Vset,
         sem   := LAMBDA s0, s1, o :
                    CompInterface(Eset,Vset,s0,s1,o) AND
                    NonTerm(s0,s1) AND TimeAdvance(s0,s1) AND
                    (A(s0,o) IMPLIES C(s1,o))
       #)
```

Finally we define a conversion `At` which allows us to consider observation functions as a function from events to predicates over `Time`. Then we can use `o(E)`, which is interpreted as `At(o)(E)`, as a predicate over time and apply the notations of theory `RTPrim` to it.

```
At(o)(E)(t) : bool  = o(t)(E)
CONVERSION At
END Specs
```

Theories can be parsed and type checked. Since type correctness in PVS is undecidable, type checking might generate Type Check Conditions which should be proved by the user. In case of theory `Specs` there is one Type Check Condition for the definition of `spec`. It requires a proof that the definition indeed has the type `Components`, that is, satisfies constraint `CompProp`, which is trivial here.

Example

Theory `SpecsExs` contains a few simple examples to illustrate the definitions above. Suppose the set of variables is defined by an enumeration type {`x,y,z`} and the set of events by {`read,write`}. Elements of an enumeration type are different in PVS, thus, e.g., `read /= write`. First observe that a commitment which asserts the occurrence of an event outside the alphabet is equivalent to false, as illustrated by lemma `PEmptyLem`.

```
SpecsExs : THEORY
BEGIN
Vars   : TYPE = {x,y,z}
Events : TYPE = {read,write}
IMPORTING Specs[integer,Vars,Events]

PEmptyLem : LEMMA
  spec({E|E = read}, {V|V = x}, true, LAMBDA s,o : o(7)(write))
  =
  spec({E|E = read}, {V|V = x}, true, false)
```

Next consider specification `P1` with local variable `x` and a similar specification `P2` which additionally expresses that the value of `y` remains `0`. Lemma `P1refP2` expresses that this property follows from `P1`. This lemma can be proved automatically (using property `CompInterface`) by the command `grind`, but a more interactive proof, where the user provides the proper commands, requires less time. This reflects a general observation, namely that more user interaction often leads to more efficient proofs.

```
P1 : Components =
     spec({E|E = read OR E = write}, {V|V = x},
           LAMBDA s, o : term(s) AND now(s) = 2,
           LAMBDA s, o : dur(o(read), co(3,4)) AND
                         inside(o(write), co(6,8)) AND
                         term(s) AND now(s) = 10 AND
                         val(s)(x) = 7 )
P2 : Components =
    spec({E|E = read OR E = write}, {V|V = x},
          LAMBDA s, o : term(s) AND now(s) = 2 AND val(s)(y) = 0,
          LAMBDA s, o : dur(o(read), co(3,4)) AND
                        inside(o(write), co(6,8)) AND
                        term(s) AND now(s) = 10 AND
                        val(s)(x) = 7 AND val(s)(y) = 0 )
P1refP2 : LEMMA P1 => P2
```

3 Semantics Parallel Composition

In this section we define the semantics of parallel composition. In general, there are several possibilities to define programming constructs in PVS. One approach is to start with a formalization of the syntax of programming constructs and then define a semantic function which maps the syntactic structure into a semantic domain. To avoid deep encodings and to have more flexibility to mix programs and specifications, we follow here the alternative approach (as in [Hoo94b]) and identify programming constructs and their semantics.

The semantics of parallel composition is defined in theory `SemParComp`. First `par(comp1,comp2)` is defined, requiring that the sets of variables of the two components are disjoint. The sets of events might overlap, with the intention that the components synchronize on these joint events. The states of the components are related using predicate `CombineStates` which expresses that the values of the variables of both components are combined and the maximum of the termination times is taken. This is used in `StateRel` which additionally expresses that a parallel component terminates iff both components terminate.

```
SemParComp [Values,Vars, Events : TYPE] : THEORY
BEGIN
IMPORTING Specs[Values,Vars,Events]

disjunct(Vset1,Vset2) : bool =
                         intersection(Vset1,Vset2) = emptyset
disjunct(comp1,comp2) : bool = disjunct(vari(comp1),vari(comp2))
```

```
CombineVals(Vset1,(Vset2 | disjunct(Vset1,Vset2)),
            s0,s1,s2) : [Vars -> Values] =
               LAMBDA V : COND member(V, Vset1) -> val(s1)(V),
                                member(V, Vset2) -> val(s2)(V),
                          ELSE                  -> val(s0)(V)
                          ENDCOND

CombineStates(Vset1,(Vset2 | disjunct(Vset1,Vset2)),
              s0,s1,s2) : State =
                 (# val  := CombineVals(Vset1,Vset2,s0,s1,s2),
                    now  := max(now(s1),now(s2)),
                    term := true
                  #)

StateRel(Vset1,(Vset2 | disjunct(Vset1,Vset2)),
         s0,s1,s2,s) : bool =
                  IF term(s1) AND term(s2)
                  THEN s = CombineStates(Vset1,Vset2,s0,s1,s2)
                  ELSE NOT term(s) ENDIF;
```

Next parallel composition is defined, taking the union of the observation functions provided they agree (i.e., are equal) on the joint events of both components (i.e., the events that are in both alphabets).

```
par(comp1,(comp2 | disjunct(comp1,comp2))) : Components =
  (# alpha := union(alpha(comp1),alpha(comp2)),
     vari  := union(vari(comp1),vari(comp2)),
     sem   := (LAMBDA s0, s, o : NonTerm(s0,s) AND
               (EXISTS s1, s2, o1, o2 :
                sem(comp1)(s0,s1,o1) AND sem(comp2)(s0,s2,o2) AND
                StateRel(vari(comp1),vari(comp2),s0,s1,s2,s) AND
                o = union(o1,o2) AND
                intersection(o1,
                    intersection(alpha(comp1),alpha(comp2))) =
                intersection(o2,
                    intersection(alpha(comp1),alpha(comp2))) ) )
   #)
```

Since the result of the parallel composition should be of type `Components`, type checking leads to the a Type Check Condition which requires a proof that the record in the definition indeed satisfies `CompProp`.

Given both the importance and the complexity of this definition, we would like to increase our confidence, i.e. check whether it corresponds to the intuition. This can be done by means of examples, as shown later, and by proving theorems about parallel composition. For instance, commutativity and associativity have been proved. Further we have proved that if the alphabets are disjoint, that is, `intersection(alpha(comp1),alpha(comp2)) = emptyset`, then the clause

```
intersection(o1,intersection(alpha(comp1),alpha(comp2))) =
intersection(o2,intersection(alpha(comp1),alpha(comp2)))
```

can be removed from the definition of `par(comp1,comp2)`.

Moreover we formulate an alternative definition and show that it is equivalent to the definition above. We use the infix operator `//`, which means that we can write `comp1 // comp2`. The difference with the previous definition is the treatment of the observation functions. Similar to the definition for traces in [Hoa85], we require that the projection of an observation function of the parallel construct onto the alphabet of one of the components leads to an observation of this component. Additionally, any observation function of the composition should only contain events of its alphabet.

```
// (comp1, (comp2 | disjunct(comp1,comp2)) ) : Components =
  (# alpha := union(alpha(comp1),alpha(comp2)),
     vari  := union(vari(comp1),vari(comp2)),
     sem   := (LAMBDA s0, s, o : NonTerm(s0,s) AND
               (EXISTS s1, s2, o1, o2 :
                sem(comp1)(s0,s1,o1) AND sem(comp2)(s0,s2,o2) AND
                StateRel(vari(comp1),vari(comp2),s0,s1,s2,s) AND
                intersection(o,alpha(comp1)) = o1 AND
                intersection(o,alpha(comp2)) = o2 AND
                subset?(o, union(alpha(comp1),alpha(comp2))) ) )
   #)
```

Then we can prove that the two definitions are equivalent.

```
AltSemEquiv : THEOREM
     FORALL comp1, (comp2 | disjunct(comp1,comp2)) :
                             comp1//comp2 = par(comp1,comp2)
```

Example

To illustrate the requirement on the local variables, suppose a theory contains the parallel composition `P1 // P2` of the components `P1` and `P2` defined before. Then type checking requires the proof of the Type Check Condition

```
ParP1P2_TCC1: OBLIGATION disjunct(P1, P2)
```

This cannot be proved because `x` is a variable of both `P1` and `P2`.

4 Proof Rules

The refinement expressed by lemma `P1refP2` of section 2 has been proved by unfolding all semantic definitions. Similarly, refinements involving parallel composition could be proved on the basis of the semantics. Especially for complex systems this is rather inconvenient and hence proof rules based on specifications are formulated. First, section 4.1 contains a version of the classical consequence rule. The next two sections deal with parallel composition. The aim is to obtain a rule where the commitment of a parallel composition is essentially the conjunction of the commitments of the components, inspired by similar rules for untimed systems (see, e.g., [Old85, Zwi89]).

4.1 The Consequence Rule

To allow weakening of commitments and strengthening of assumptions, the usual consequence rule is proved as a theorem in PVS.

```
ParRules [Values, Vars, Events : TYPE] : THEORY
BEGIN
IMPORTING SemParComp[Values,Vars,Events]

ConsRule : THEOREM Eset = Eset0 AND Vset = Vset0 AND
           Valid(A IMPLIES A0) AND Valid(C0 IMPLIES C)
           IMPLIES
           (spec(Eset0, Vset0, A0, C0) => spec(Eset, Vset, A, C))
```

4.2 A Simple Rule for Parallel Composition

A simple, but often very useful, rule for parallel composition can be formulated if validity of the commitments is independent of the state. Predicate `StateIndep(A)` expresses that assertion `A` does not depend on the state, i.e., its validity is independent of the values of `now`, `term`, and the variables.

```
StateIndep(A) : bool = FORALL s1, o :
                         A(s1,o) IMPLIES (FORALL s2 : A(s2,o))
```

To obtain a sound rule it is additionally required that the validity of assertions of a component only depends on the interface of this component. Condition `OnlyDepEve(A,Eset)` expresses that validity of assertion `A` only depends on the events in `Eset`.

```
OnlyDepEve(A,Eset) : bool = FORALL s, o :
                              A(s,o) IFF A(s,intersection(o,Eset))
```

This leads to parallel composition rule `RulePar`.

```
RulePar : THEOREM
  intersection(Vset1,Vset2) = emptyset AND
  StateIndep(C1) AND StateIndep(C2) AND
  OnlyDepEve(A1,Eset1) AND OnlyDepEve(C1,Eset1) AND
  OnlyDepEve(A2,Eset2) AND OnlyDepEve(C2,Eset2)
  IMPLIES
  spec(Eset1,Vset1,A1,C1) // spec(Eset2,Vset2,A2,C2)
  =>
  spec(union(Eset1,Eset2),union(Vset1,Vset2),A1 AND A2,C1 AND C2)
```

To give an outline of the proof of this theorem, consider a triple `(s0,s,o)` in the semantics of the parallel composition. Then there exist `s1, o1, s2, o2` such that `(s0,s1,o1)` is in the semantics of the first component and `(s0,s2,o2)` in that of the second with e.g. `intersection(o,Eset1) = o1`. The main point is to show that `(A1 AND A2)(s0,o)` implies `(C1 AND C2)(s,o)`. By symmetry it is sufficient to show `C1(s,o)`. Basic idea is that `A1(s0,o)` leads by assumption `OnlyDepEve(A1,Eset1)` to `A1(s0,intersection(o,Eset1))`, that is, `A1(s0,o1)`. Using the specification of the first component, we obtain `C1(s1,o1)`. Then `OnlyDepEve(C1,Eset1)` implies `C1(s1,intersection(o1,Eset1))`, thus `C1(s1,intersection(o,Eset1))`, and again using `OnlyDepEve(C1,Eset1)` we obtain `C1(s1,o)`. Finally, `StateIndep(C1)` leads to `C1(s,o)`.

4.3 Example Parallel Composition

As an application of `RulePar`, we consider a simplified version of a mine pump example [J96]. Consider a mine where the presence of gas could lead to an explosion. No local variables are needed and `Vars` is declared as an uninterpreted type. The top-level specification of the system, `TLComp`, expresses that the system starts at time `0` (assumption `A`) and that there should be no explosion (commitment `CTL`).

```
ExplEx : THEORY
BEGIN
Vars   : TYPE
Events : TYPE = {expl,gas}
IMPORTING ParRules[nat,Vars,Events]

ETL : setof[Events] = {E|E = expl OR E = gas}
A   : Assertion = LAMBDA s, o : term(s) AND now(s) = 0
CTL : Assertion = LAMBDA s, o : (FORALL t : NOT o(t)(expl))

TLComp : Components = spec(ETL,emptyset,A,CTL)
```

Suppose it is known that there can only be an explosion in the mine if gas is present during at least `CritGasPeriod` time units. This leads to the following specification of the mine component.

```
EM : setof[Events] = {E | E = expl OR E = gas}

CritGasPeriod : Time

CM : Assertion = LAMBDA s, o : (FORALL t :
      o(t)(expl) IMPLIES
      t >= CritGasPeriod AND dur(o(gas), cc(t-CritGasPeriod,t)) )

MComp : Components = spec(EM,emptyset,A,CM)
```

To achieve the top-level specification, we specify a control component which should ensure that if there is gas in the mine then the gas is removed within at most `Delay` time units.

```
EC : setof[Events] = {E | E = gas}

Delay    : Time       % delay to remove gas

CC : Assertion = LAMBDA s, o : (FORALL t :
          o(t)(gas) IMPLIES inside(NOT o(gas), oc(t,t+Delay)) )

CComp : Components = spec(EC,emptyset,A,CC)
```

Next we show that `MComp // CComp` leads to the top-level specification. The conditions of the parallel composition rule are easy to prove. The main point of the proof is given by lemma `CommitLem` which expresses that the commitments of the mine and the control component lead to the top-level commitment.

```
CommitLem : LEMMA Delay <= CritGasPeriod IMPLIES
                  Valid(CM AND CC IMPLIES CTL)

TLRef : THEOREM Delay <= CritGasPeriod
                IMPLIES (MComp // CComp => TLComp)
END ExplEx
```

4.4 Other Rules for Parallel Composition

We present two other rules for parallel composition. The first rule expresses that a parallel composition does not terminate if one of the components does not terminate. Since parallel composition is commutative, we can assume that the first component does not terminate.

```
RuleParNonTerm : THEOREM
  intersection(Vset1,Vset2) = emptyset AND
  StateIndep(C1) AND StateIndep(C2) AND
  OnlyDepEve(A1,Eset1) AND OnlyDepEve(C1,Eset1) AND
  OnlyDepEve(A2,Eset2) AND OnlyDepEve(C2,Eset2)
  IMPLIES
  spec(Eset1,Vset1,A1,C1 AND NOT term) // spec(Eset2,Vset2,A2,C2)
  =>
  spec(union(Eset1,Eset2), union(Vset1,Vset2),
       A1 AND A2, C1 AND C2 AND NOT term)
```

The next rule assumes that both components terminate. Then also the states of the two components have to be related. As expressed by `Combine(C1,C2,C)`, we can again take the conjunction of the commitments except for the value of `now` which becomes the maximum of the two termination times. Clearly, the commitment of a component should be restricted to the local variables of the component. Condition `OnlyDepVar(A,Vset)` expresses that validity of `A` only depends on variables in `Vset`.

```
Combine(C1,C2,C) : bool = FORALL s, o, t1, t2 :
   C1(s WITH [now := t1],o) AND C2(s WITH [now := t2],o)
   IMPLIES
   C(s WITH [now := max(t1,t2)],o)

OnlyDepVar(A,Vset) : bool = FORALL s1, s2, o:
  A(s1,o) AND
  (FORALL V : member(V,Vset) IMPLIES val(s1)(V) = val(s2)(V))
  IMPLIES
  A(s2 WITH [now := now(s1), term := term(s1)],o)

RuleParTerm : THEOREM
 intersection(Vset1,Vset2) = emptyset AND
 OnlyDepEve(A1,Eset1) AND OnlyDepEve(C1,Eset1) AND
 OnlyDepEve(A2,Eset2) AND OnlyDepEve(C2,Eset2) AND
 OnlyDepVar(C1,Vset1) AND OnlyDepVar(C2,Vset2) AND
 Combine(C1,C2,C)
 IMPLIES    spec(Eset1,Vset1,A1,C1 AND term) //
            spec(Eset2,Vset2,A2,C2 AND term)
            =>
            spec(union(Eset1,Eset2), union(Vset1,Vset2),
                 A1 AND A2, C AND term)
END ParRules
```

5 Asynchronous Communication

The theories above have been formulated using a general set of events. Given a particular communication mechanism, the set of events can be more structured. This is illustrated here by a formalization of asynchronous value passing along unidirectional channels. First we define a general theory `Asyn`, with a given a set of channels as parameter. Events are defined to express sending and receiving, and to express that a process is waiting to receive (when no value is available). We introduce events for communication with and without values and give two axioms to relate these events. Finally axiom `ValueAx` expresses that at most one value is transmitted at any point in time.

```
Asyn [Values, Vars, Chan : TYPE] : THEORY
BEGIN
ChanEvents : NONEMPTY_TYPE
IMPORTING ParRules[Values,Vars,ChanEvents]
ch              : VAR Chan
v, v1 , v2      : VAR Values
```

```
sendv(ch,v) : ChanEvents
send(ch)    : ChanEvents
recv(ch,v)  : ChanEvents
rec(ch)     : ChanEvents
waitrec(ch) : ChanEvents

AbstrSendAx : AXIOM o(send(ch))(t) IFF EXISTS v : o(sendv(ch,v))(t)

AbstrRecAx : AXIOM o(rec(ch))(t) IFF EXISTS v : o(recv(ch,v))(t)

ValueAx : AXIOM o(sendv(ch,v1))(t) AND o(sendv(ch,v2))(t)
                IMPLIES v1 = v2
END Asyn
```

Next we consider in section 5.1 the case that values are not buffered. An axiomatization of buffered communication is presented in section 5.2.

5.1 Asynchronous Communication without Buffering

Suppose values are not buffered; a sender can send its value immediately along a channel and if the receiver is not ready to receive it, then this value gets lost. A receiver might have to wait until a value is available. Assuming, for simplicity, that value transmission does not take any time, axiom `MinWaitAx` expresses that it should not be possible that a process is sending and simultaneously (another) process is waiting to receive. Further, axiom `RecSendAx` states that if a value is received then it is sent simultaneously (since we ignore transmission time and assume there is no buffering). Based on these axioms, a number of lemmas has been proved. Here we only show lemma `ValsrLem`.

```
AsynNoBuf [Values, Vars, Chan : TYPE] : THEORY
BEGIN
IMPORTING Asyn[Values,Vars,Chan]

MinWaitAx : AXIOM NOT (o(send(ch))(t) AND o(waitrec(ch))(t))

RecSendAx : AXIOM o(recv(ch,v))(t) IMPLIES o(sendv(ch,v))(t)

ValsrLem  : LEMMA o(sendv(ch,v1))(t) AND o(recv(ch,v2))(t)
                  IMPLIES v1 = v2
```

As an abbreviation, we introduce `awaitrec(ch,o)` to express that a process is waiting to receive until a value is received, i.e., either the process waits forever (because no value becomes available), or it eventually receives a value.

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

Reasoning with the axioms above is rather cumbersome and it is more convenient to formulate theorems for certain situations. As an example, we show that no value gets lost if the sender does not send too often and the receiver is ready to receive early enough to receive the next value.

```
Init, Period    : VAR Time

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

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

SendRecvLem : LEMMA
    (FORALL t : maxsend(ch,Init,Period,o)(t)) AND
    (FORALL t : minawait(ch,Init,Period,o)(t))
     IMPLIES
    (FORALL t, v : o(sendv(ch,v))(t) IFF o(recv(ch,v))(t))
```

To shorten the description of the interface of components, we introduce notations to transform a set of channels into a set of corresponding input or output events.

```
Cset : VAR setof[Chan]
che  : VAR ChanEvents

InEve(Cset) : setof[ChanEvents] = LAMBDA che :
 (EXISTS ch : member(ch,Cset) AND (che = waitrec(ch) OR
     (EXISTS v : che = recv(ch,v)) OR che = rec(ch) ) )

OutEve(Cset) : setof[ChanEvents] = LAMBDA che :
 (EXISTS ch : member(ch,Cset) AND
     ((EXISTS v : che = sendv(ch,v)) OR che = send(ch) ) )
END AsynNoBuf
```

Example

We apply this theory to a simple example of real-time components. First we formulate the top-level specification which expresses that after receiving a value `v` along channel `datain`, the value `f2(f1(v))` is transmitted along `dataout` in less than `deadline` time units. Since hiding has not yet been introduced, internal channel `internal` is included in the set of events of the system. Also the variables `x` and `y` are already introduced.

```
ParEx : THEORY
BEGIN
Values : TYPE = real
```

```
Vars   : TYPE = {x,y}
Chan   : TYPE = {datain,internal,dataout}
IMPORTING AsynNoBuf[Values,Vars,Chan]
f1, f2 : [Values -> Values]

ChanSet : setof[ChanEvents] =
          union(InEve({ch|ch = datain OR ch = internal}),
                OutEve({ch|ch = internal OR ch = dataout}))

VarSet : setof[Vars] = {V|V = x OR V = y}

A : Assertion = LAMBDA s, o : term(s) AND now(s) = 0

deadline : Time

C : Assertion = LAMBDA s, o : (FORALL t, v :
   o(recv(datain,v))(t) IMPLIES
    inside( o(sendv(dataout, f2(f1(v)) )) , co(t,t+deadline) ))

F1F2Comp : Components = spec(ChanSet,VarSet,A,C AND NOT term)
```

The idea is to implement this by two parallel components, where the first one computes function `f1` and sends this value along channel `internal` to the second component which then computes `f2`. Since no value should get lost, we specify - in view of lemma `SendRecvLem` - the maximum sending frequency of the first component along channel `internal`.

```
ChanSet1 : setof[ChanEvents] =
    union(InEve(singleton(datain)), OutEve(singleton(internal)))

VarSet1 : setof[Vars] = {V|V = x}

deadline1, Init, Period : Time

C1 : Assertion = LAMBDA s, o :
      (FORALL t : maxsend(internal,Init,Period,o)(t)) AND
      (FORALL t, v : o(recv(datain,v))(t) IMPLIES
         inside( o(sendv(internal, f1(v))), co(t,t+deadline1) ))

F1Comp : Components = spec(ChanSet1,VarSet1,A,C1 AND NOT term)
```

Similarly, the second component is specified, now expressing that it is ready to receive along `internal` periodically.

```
ChanSet2 : setof[ChanEvents] =
    union(InEve(singleton(internal)), OutEve(singleton(dataout)))
```

```
VarSet2 : setof[Vars] = {V|V = y}

deadline2 : Time

C2 : Assertion = LAMBDA s, o :
      (FORALL t : minawait(internal,Init,Period,o)(t)) AND
      (FORALL t, v : o(recv(internal,v))(t) IMPLIES
         inside( o(sendv(dataout, f2(v))), co(t,t+deadline2) ))

F2Comp : Components = spec(ChanSet2,VarSet2,A,C2)
```

Then, using lemma `SendRecvLem`, we can prove the main step of the refinement as represented by lemma `F1F2CLem`.

```
F1F2CLem : LEMMA deadline1 + deadline2 <= deadline IMPLIES
                 Valid(C1 AND C2 IMPLIES C)

F1F2Ref : THEOREM deadline1 + deadline2 <= deadline
                  IMPLIES (F1Comp // F2Comp => F1F2Comp)
END ParEx
```

5.2 Asynchronous Communication with Buffering

In this section, asynchronous communication with buffering is modelled. First assume that buffers are infinite. The idea is to express that the n-th receive event of a channel corresponds to the n-th send event. To be able to count the number of occurred events at a certain point in time, we assume starting time 0 and finite variability, that is, only a finite number of events occur in any finite amount of time. To express this in PVS we import a library with theories about finite sets, including function `card` which yields the cardinality of a finite set.

```
AsynBuf [Values, Vars, Chan : TYPE] : THEORY
BEGIN
IMPORTING Asyn[Values,Vars,Chan]
NonNegTime : TYPE = { t : Time | t >= 0 }
t, t0, t1, t2   : VAR NonNegTime

fsets: LIBRARY = "../pvs/lib/new_finite_sets"
IMPORTING fsets@finite_sets[NonNegTime]
n : VAR nat

OccurrencePoints(che,o)(t) : setof[NonNegTime] =
                             LAMBDA t1 : t1 <= t AND o(t1)(che)

finite_variability : AXIOM is_finite(OccurrencePoints(E,o)(t))

OccNumber(che,o,t) : nat = card(OccurrencePoints(che,o)(t))
```

Axiom `BufMinWait` asserts that a process can only wait to receive a value if the buffer is empty, that is, the number of values sent equals the number of values received. Axiom `BufRecSend` expresses the required correspondence between values.

```
BufMinWait : AXIOM o(waitrec(ch))(t) IMPLIES
                   OccNumber(send(ch),o,t) = OccNumber(rec(ch),o,t)

BufRecSend : AXIOM o(recv(ch,v))(t) IMPLIES
   EXISTS n, t1 : OccNumber(recv(ch,v),o,t) = n AND t1 <= t AND
                  o(sendv(ch,v))(t1) AND
                  OccNumber(sendv(ch,v),o,t1) = n
END AsynBuf
```

The theory above can be extended easily to model bounded buffers, say for buffer size `N`. Then additionally it has to be specified what happens when the buffer is full. For instance, the value transmitted can be ignored or one of the buffer elements could be deleted. Here we specify that the sender should wait until the receiver has removed an element from the buffer. Hence an additional event is introduced to model this waiting behaviour. Axiom `MinWaitSend` expresses that waiting to send is only allowed if the buffer is full.

```
AsynBoundBuf [N : nat, Values, Vars, Chan : TYPE] : THEORY
BEGIN
IMPORTING AsynBuf[Values,Vars,Chan]

waitsend(ch)   : ChanEvents

MinWaitSend : AXIOM o(waitsend(ch))(t) IMPLIES
              OccNumber(send(ch),o,t) = OccNumber(rec(ch),o,t) + N
END AsynBoundBuf
```

6 Concluding Remarks

Other applications of PVS often concern the use of this tool during the early phases of system design, when the requirements are formulated and formalized [Rus95]. This is motivated by the fact that most errors are often due to mistakes in the requirements specification and high-level design. To investigate a requirements specification one can try to prove so-called putative theorems stating properties that ought to hold in the system. This could concern expected consequences of the specification or the equivalence with alternative formulations. Especially failures to prove certain theorems are often useful, since it points to errors or misunderstandings.

Although this paper mainly concerns the development of a formal theory, there are certain similarities with system design. Also here attempts to prove properties revealed many errors. We present a few examples.

– Attempts to prove the Type Check Condition for the `par` operator showed errors in the definition of `CompProp` (e.g., `CompInterface` only has to hold in case of termination) and in the definition of the semantics (e.g., `NonTerm` has been added). In general, the type checking mechanism of PVS frequently reveals errors.
– An attempt to prove equivalence of the semantic formulations of parallel composition (lemma `AltSemEquiv`) showed an error in the combination of the observation functions.
– A first version of the consequence rule, allowing subsets of events and local variables turned out to be unsound.
– An error in the semantics of parallel composition was found when trying to prove the simple parallel composition rule (a case distinction based on `term(s0)` has been removed).
– Unsoundness of a complicated version of the parallel composition rule was detected when trying to prove it.

The PVS proof checker contains quite powerful proof commands, but most proofs still require a rather intensive user interaction. Especially for complex proofs, it is important to have a clear idea about the main steps of a proof, and to introduce suitable lemmas. Given the fact that during the development of a theory many changes are made, it is very convenient that proofs are stored in PVS and can be rerun to check which proofs have to be adapted.

Concerning the work described here, there are several directions for future research. The sequential programming constructs that have been formalized earlier [Hoo94b] should be incorporated in the current framework. Then it is relevant to make a comparison with related work, such as the formalization of the refinement calculus in HOL [Bn96], although that work does not deal with real-time and parallelism. In other work we will report about the addition of a hiding construct to encapsulate internal channels. More complex examples should indicate other required adaptations and extensions. Further, applications might require the axiomatization of other communication mechanisms such as synchronous communication or global variables.

Acknowledgement

Thanks are due to an anonymous reviewer for valuable suggestions for improvement.

References

[Bn96] M. Butler and T. Långbacka. Program derivation using the refinement calculator. In *Theorem Proving in Higher Order Logics*, pages 93–108. LNCS 1125, Springer-Verlag, 1996.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, 1969.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hoo91] J. Hooman. *Specification and Compositional Verification of Real-Time Systems.* LNCS 558, Springer-Verlag, 1991.

[Hoo93] J. Hooman. A compositional approach to the design of hybrid systems. In *Workshop on Theory of Hybrid Systems*, pages 121–148. LNCS 736, 1993.

[Hoo94a] J. Hooman. Compositional verification of a distributed real-time arbitration protocol. *Real-Time Systems*, 6(2):173–205, 1994.

[Hoo94b] J. Hooman. Correctness of real time systems by construction. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 19–40. LNCS 863, Springer-Verlag, 1994.

[Hoo95] J. Hooman. Verifying part of the ACCESS.bus protocol using PVS. In *Proceedings 15th Conference on the Foundations of Software Technology and Theoretical Computer Science*, pages 96–110. LNCS 1026, Springer-Verlag, 1995.

[Hoo96] J. Hooman. Using PVS for an assertional verification of the RPC-memory specification problem. In *Formal Systems Specification; The RPC-Memory Specification Case Study*, pages 275–304. LNCS 1169, Springer-Verlag, 1996.

[Hoo97] J. Hooman. Verification of distributed real-time and fault-tolerant protocols. In *Algebraic Methodology and Software Technology (AMAST'97)*, pages 261–275. LNCS 1349, Springer-Verlag, 1997.

[J96] M. Joseph, Editor. *Real-time Systems: Specification, Verification and Analysis.* Prentice Hall, 1996.

[Old85] E.-R. Olderog. Process theory: Semantics, specification and verification. In *Current Trends in Concurrency*, pages 442–509. LNCS 224, Springer-Verlag, 1985.

[ORS92] 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.

[ORSvH95] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, 1995.

[Rus95] J. Rushby. Mechanizing formal methods: Opportunities and challenges. In *ZUM '95: The Z Formal Specification Notation; 9th International Conference of Z Users*, pages 105–113. LNCS 967, Springer-Verlag, 1995.

[SO98] M. Schenke and E.-R. Olderog. Transformational design of real-time systems – Part I: From requirements to program specification. *Acta Informatica*, to appear, 1998.

[VH96] J. Vitt and J. Hooman. Assertional specification and verification using PVS of the steam boiler control system. In J.-R. Abrial, E. Börger, and H. Langmaack, editors, *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, pages 453–472. LNCS 1165, Springer-Verlag, 1996.

[Zwi89] J. Zwiers. *Compositionality, Concurrency and Partial Correctness.* LNCS 321, Springer-Verlag, 1989.

This article was processed using the LaTeX macro package with LLNCS style