Modular Formal Specification of Data and Behaviour

Jaco van de Pol[1], Jozef Hooman[2], Edwin de Jong[3]

[1] Dept. of Computing Science, Eindhoven University of Technology
P.O. Box 513, 5600 MB, Eindhoven, The Netherlands
jaco@win.tue.nl

[2] Computing Science Institute, University of Nijmegen
P.O. Box 9010, 6500 GL, Nijmegen, The Netherlands
hooman@cs.kun.nl

[3] Hollandse Signaalapparaten B.V.
P.O. Box 42, 7550 GD, Hengelo, The Netherlands
edejong@signaal.nl

Abstract. We propose a modular approach to the formal specification of the requirements on embedded systems. In this approach, requirements on data are specified as invariants on states. Requirements on behaviour are specified assertionally by temporal logic formulae, restricting the runs of the system. The proposed method is modular, because components can be specified and analysed in isolation, and the views of several components can be combined in an easy way. Requirements can be combined by simply putting them in conjunction. A mathematical framework supporting this approach is developed and implemented in the theorem prover PVS. The method is illustrated by formalising the requirements of a miniature embedded system. This specification is then analysed using the theorem prover, revealing some errors in the original specification.

Keywords: formal requirements specification, embedded systems, modularity, data and behaviour, states and events, linear temporal logic, PVS.

1 Introduction

The requirements specification is the first formal document in the development of a complex system. Errors in the specification propagate to all later phases in the development, until they are detected. It is well-known that errors that are made early and detected late, are relatively expensive to repair [16, p. 38]. Hence, the quality of a requirements specification is important. Good analysis methods are helpful to improve this quality.

The requirements specification cannot be verified against some prior formal document. Besides some well-formedness checks, it can only be validated against informal user requirements. Various validation techniques have been developed to assess the quality of specifications, like inspection, prototyping and scenario generation.

Still formal methods are helpful in this early phase of system development, cf. [16]. Parsing and type checking a specification reveal numerous errors. The formal semantics of the specification language provides a mathematical model of the specified system, which can be analysed with mathematical rigour. Properties of the specification, like consistency, can be verified formally and validation is supported by proving formal challenges. Powerful theorem provers are available to make this analysis tractable for large systems.

The contribution of this paper is a framework to specify requirements on data and on behaviour of complex systems in a modular fashion. In order to have tool support at this experimental stage, we use an existing general-purpose theorem prover. We have experimented [14] with PVS (Prototype Verification System) [12, 13], which is based on typed higher-order classical logic.

The basic framework. We distinguish between states and events. The state represents the data in the system, that has been deduced about the current state of affairs in the environment. Events occur instantaneously on the border of the system. Events can be split into input and output events. An event causes a state transition, which also takes place instantaneously. A *run* of the system is an infinite sequence $s_0, e_0, s_1, e_1, s_2, \ldots$, where the s_i are states and the e_i are events. A run represents one possible behaviour of the system. In this approach, simultaneous events are modelled by all possible interleavings.

Such systems could be specified by transition systems. However, in order to obtain more abstract specifications, we use assertions on states and on runs. Assertions on states represent relationships between data items that should be maintained invariantly. By allowing invariants, we don't need to specify *how* the constraints shall be maintained. Assertions on runs will specify the behaviour of the system. In this paper we will restrict such assertions to formulae of linear temporal logic, based on propositions on states, state transitions and events.

Modularisation. To achieve readability and scalability, some structure has to be imposed on large specifications. To this end, the specification of the system is decomposed into a specification of components. These components must not be understood as structural or physical parts of the system to be constructed; this decomposition is deferred to the design phase. The components can be understood as projections of the system, focusing on a number of state variables and a number of events. The requirements on a component are specified in terms of its own state variables and events only.

Consider subcomponents C1 and C2 of some aggregate A. C1 and C2 are specified completely independently and don't even know about each other's existence. We don't specify internal communication between C1 and C2. But, because C1 and C2 are subcomponents of A, requirements on A may use their variables and events, so coordination between C1 and C2 can be specified at the level of A.

Problem statement. Our aim is to develop a formal framework for the requirements specification of embedded systems. To capture the requirements, the method should allow assertional specifications of data and behaviour. For scalability, the method should be modular, by allowing that components are specified locally, independent of the complete system. Coordination between various components is specified separately.

A technical problem that has to be solved concerns the interpretation of local assertions of a component on a global level. It should be possible to put the requirements of different components in conjunction. We look for a solution that is easily implementable in PVS.

The rest of this paper. Section 2 informally introduces a running example. The implementation in PVS consists of a generic part (Section 3) and a system specific part (Section 4). The generic part contains the definition of temporal logic and its modular interpretation. In the specific part, we formalise the requirements of the running example, using the generic framework. Section 5 reports on the analysis of a requirements specification. The results are summarised in Section 6, in which also related and future work are mentioned.

2 Running Example

Embedded systems are equipped with various sensors and actuators. Measurements from the environment are continuously obtained via the sensors and compiled into an abstract picture that reflects the current state of the environment. This picture is communicated to a team of operators. The system supports the decision making process by tracking differences between the perceived state and the required state, and by proposing and analysing corrective actions, which can be executed via the system's actuators. Applications of these systems include traffic management systems, command and control systems, and process control systems.

We now introduce a running example which can be seen as a miniature embedded system. The system interacts with two different sensors, and with an operator. For simplicity it is assumed that the sensors only report *increment* events, that are counted by the system. Apart from the *value* of these two counters, there is a derived data item, z, which represents the difference between these two values.

The system interactively diagnoses the interpreted information and communicates with the operator, via a warning mechanism. If this mechanism is *active*, a warning shall be *raised*, whenever the value of z exceeds 10. The operator can *close* a raised warning. When z is lower than or equal to 10 again, the system may also *withdraw* the warning. Finally, the operator can *toggle* the activity of the warning system.

Any decomposition of this small system is artificial, but to illustrate our approach, we divide the system into a warning component and a database component. The database contains two counters as subcomponents, that hold

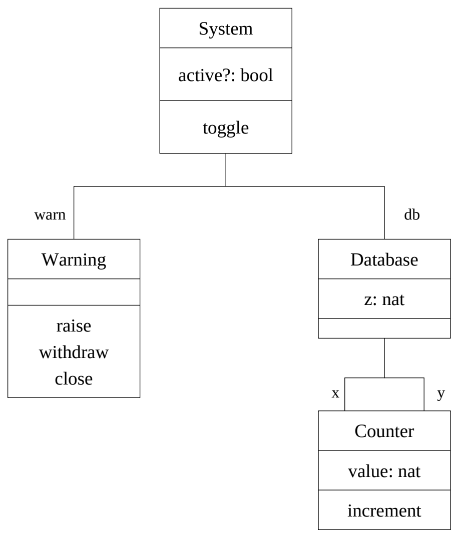


Fig. 1. Hierarchic structuring of running example

the values of the corresponding sensors. Figure 1 gives an impression of the structure thus obtained. Moreover, this figure assigns names to the sub-components (like `x` and `db`), distributes typed state variables (e.g. `z:nat`, `active?:bool`) over components, and also distributes the events (e.g. `raise`, `toggle`) of the system over the components.

We now illustrate how the requirements can be specified by assertions that are local to components. We stress that the real specification is the formal specification in Section 4.

The warning component has no explicit state. Only the possible orders of event occurrences have to be specified. The fact that two raise-events must be interleaved by a withdraw or close event, is expressed as a temporal logic formula:

$$\Box(raise \Rightarrow \bigcirc((withdraw \vee close) \mathsf{B}\, raise)),$$

where $\Box$ is read as *always*, $\bigcirc$ as *next* and B as *before*. In words: Whenever a raise occurs, the next raise should be preceded by either a withdraw or a close.

In the counter component, it must be specified what the effect of the increment event is. This effect is a state relation that can be expressed as: $value' = value + 1$, meaning that the value in the next state equals the current value incremented by one. The value of z in the database component is specified as an invariant. This can be expressed as $z = x.value - y.value$. At the system level, we have to specify the coordination between the warning and database components. Most importantly, we have to specify that a raise-event should occur in a dangerous situation: $\Box(becomes(db.z > 10) \Rightarrow \Diamond(warn.raise))$, where $\Diamond$ is read as *eventually*, and $becomes(p)$ means that p becomes true in the current state transition.

2.1 Towards a Formalisation

Composition of State Variables and Events. States and events can be modelled in a dual way. Because a component may inspect its own variables and the variables in its subcomponents, the state of each component is modelled as the Cartesian product of its own variables and the states of its subcomponents (see Figure 2). On the other hand, if an event occurs, it either is an event of the component itself, or in one of its subcomponents. Therefore, the set of events of each component can be modelled as the disjoint union of its own events, and the events of its subcomponents (see Figure 3). Disjointness is needed to distinguish e.g. the increment events of the two counters.

In the example, the database can use *x.value*. The same state variable is accessed by the system as *db.x.value*. Here *x* and *db* can be seen as projections that come for free with the Cartesian product. The system can observe *warn.raise* and *db.y.increment* as events. Note that for events, *warn*, *db* and *y* can be seen as injections, that come for free with the disjoint union.

Technical Problem Statement. The difficult step is to interpret the requirements of a component in the composed system. In Figure 4 this situation is illustrated. The solid boxes and arrows represent the state and events of a component. They exist in the context of a larger state (the dashed part) and events of other components may occur (the dashed arrows). The requirements are in terms of the solid boxes and arrows. We have to translate them to runs over the dashed boxes and arrows. For a large part, this is a contextual naming problem: *db.x.value* w.r.t. the system, actually is the same as *x.value* w.r.t. the database. But note that the local runs are also interleaved with events from other components. We found a solution that can quite elegantly be implemented in PVS.

Sketch of the solution. The solution is to define a *state mapping* from the global states to the local states, and an *event mapping* from the local events to the global events. The state mapping is a composition of the right projections, e.g. $y \circ db$. Dually, the event mapping is a composition of the right

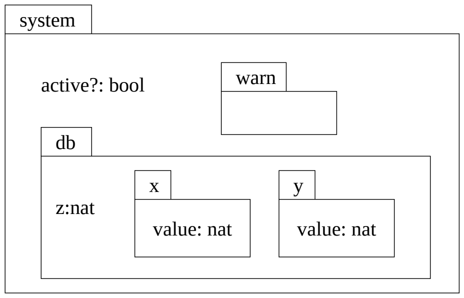


Fig. 2. The state as hierarchical Cartesian product

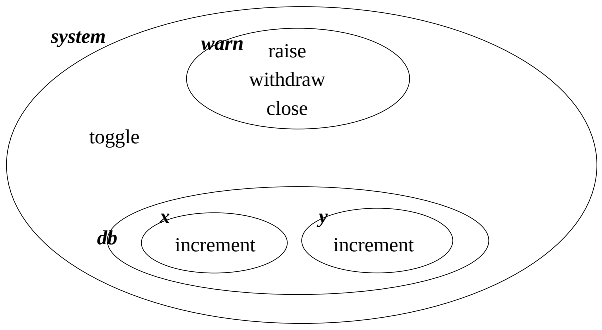


Fig. 3. The events as hierarchical disjoint union

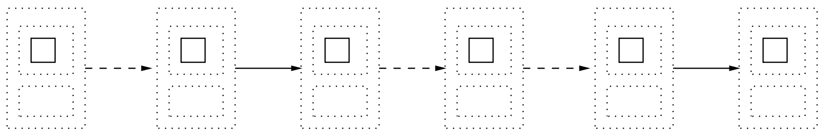


Fig. 4. Component specification in a larger context

injections, e.g. $db \circ x$. Note that multiple instances of the same component can be distinguished, because they have different state and event mappings.

Using these maps, we can interpret local requirements on runs over the global states and events! Because the requirements of all components can now be interpreted in the same global system, the requirements on the system are simply the conjunction of the requirements on all components. It remains of course to be verified that the requirements of several components are not contradictory.

3 A Framework for Requirements Specification

In our approach, a requirements specification in PVS consists of two parts. The first part, presented in this section, defines temporal logic formulae, and their modular interpretation via the above mentioned mappings. These theories are generic in the sense that they can be used in each system specification without change. The second part consists of the system specific requirements. For the running example this part is presented in Section 4.

3.1 Preliminary: the Theorem Prover PVS

PVS (Prototype Verification System [12, 13]) is a specification language and a proof checker, based on typed higher-order classical logic, in which our mathematical framework can be easily expressed. A specification can be parsed and type checked in PVS, possibly resulting in type check conditions. Besides these type check conditions, other theorems can be proved using the proof checker. When proving a theorem, PVS administrates the subgoals still to be proved. In principle, the user decides which proof rule should be applied next. PVS also provides many decision procedures for proving certain theorems automatically.

The specification language of PVS. A PVS-specification consists of a collection of *theories*, each containing a number of type and function definitions, and theorems. A theory may depend on a number of (formal) parameters for types and terms. A theory can be imported by other theories, which makes its definitions available. A number of theories are predefined, together called the prelude. This prelude contains among others basic types, e.g. natural numbers (`nat`), real numbers (`real`) and booleans (`bool`), with their usual operators. The boolean operators are written `&, =>, OR, NOT` (conjunction, implication, disjunction and negation).

We introduce some of the basic notations of PVS. For a full explanation we refer to [13]. Function types are written as in `[nat,real->bool]`, which denotes the collection of relations between natural and real numbers. Functions can be written in lambda notation, as in `LAMBDA (m:nat,z:real) : z*z=m`, which has the type just mentioned. Quantifiers can be used, as in e.g. `FORALL (m:nat): EXISTS (z:real): z*z=m`.

Record types denote Cartesian products, where the different fields are named. An example is: `[# x:nat,y:nat #]`. Terms of this type are written like `p = (# x:=3, y:=5 #)`. A record overwrite construction can be used to "change" the value of one of the fields. We could define: `q = p WITH [y := 7]`. The fields can be accessed by the names; we have `x(q)=3` and `y(q)=7`.

We will also use abstract datatypes and subtypes. Typical examples of abstract datatypes are lists and enumerated types. The syntax will be introduced later. An example of a subtype is {`z:real | EXISTS (m:nat) : z*z=m`}, which denotes the collection of square roots of natural numbers.

Lifting the booleans. As an example we give a theory to lift the usual boolean operators to predicates over arbitrary domains, which we will use several times later on (Figure 5). It is comparable to the file `connectives.pvs` in the standard PVS library.

The theory `bool2pred` is parameterised by a type, called `Domain`. The body of the theory is delimited by `BEGIN` and `END`. The body starts with variable declarations. Then some function definitions follow, extending the boolean operators pointwise to predicates. Note that overloading is allowed. The variable declarations are used to infer the type of the defined functions. For instance, `NOT` will get type `[[Domain->bool]->[Domain->bool]]`. The theory can be used for instance to lift the boolean operators to operators on binary predicates over natural numbers, by typing: `IMPORTING bool2pred[[nat,nat]]`.

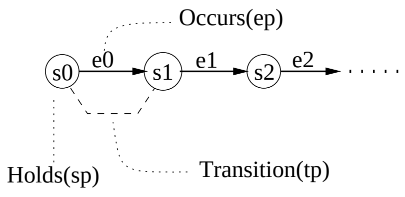
```
bool2pred[Domain:TYPE]:THEORY
BEGIN
  X: VAR Domain
  p,q: VAR [Domain->bool]

  NOT(p)(X):bool  = NOT p(X) ;
  &(p,q)(X):bool  = p(X) & q(X) ;
  OR(p,q)(X):bool = p(X) OR q(X) ;
  =>(p,q)(X):bool = p(X) => q(X)
END bool2pred
```

Fig. 5. Lifting boolean connectors to predicates

3.2 Linear Temporal Logic on Runs

Figure 6 contains a theory on linear temporal logic (cf. [10]) in PVS. The type of states and events are parameters of this theory. A run is formalised as an infinite sequence of records of states and events. A temporal formula is a predicate on runs. After some obvious variable declarations, we define three groups of temporal logic formulae.

Atomic formulae: We have three atomic formulae, expressing the fact that a predicate on states, on events, or on state transitions holds at the start of the run. A predicate on state transitions is a binary relation on successive states (called an *action* in TLA [8]):



Boolean operators: These are lifted to temporal logic by importing theory `bool2pred` of Figure 5 instantiated on runs.

Modal operators: Modal operators can be defined analogously to their usual semantic definition. `ALWAYS(p)` holds of a run, if `p` holds for all suffixes of the run (`suffix` is defined in the prelude). `EVENTUALLY` is defined as the dual of `ALWAYS`. `BEFORE(p,q)` is to be read as: `p` happens before `q` (or: `p` must precede `q`). `NEXT(p)` holds if p holds at the next moment (i.e. for the suffix obtained by deleting the first element of the run).

3.3 The Modularisation of Temporal Logic

This section explains how we can modularise temporal logic. As explained already in Section 2.1, we distinguish the local states and local events of a component, and the global states and global events of the complete system.

```
temporal_logic[State,Event: TYPE]:THEORY
BEGIN
  run: TYPE = [nat->[#state:State,event:Event#]]
  TL: TYPE  = [run->bool]

  p,q: VAR TL ;  R: VAR run ;  i,j,k: VAR nat
  sp: VAR [State->bool]         % a state predicate
  ep: VAR [Event->bool]         % an event predicate
  tp: VAR [State,State->bool]   % a transition predicate

  Holds(sp)(R):bool = sp(state(R(0)))
  Occurs(ep)(R):bool = ep(event(R(0)))
  Transition(tp)(R) :bool = tp(state(R(0)),state(R(1)))

  IMPORTING bool2pred[run]

  ALWAYS(p)(R):bool   = FORALL j: p(suffix(R,j))
  EVENTUALLY(p):TL = NOT ALWAYS(NOT p)
  BEFORE(p,q)(R):bool
    = FORALL j: q(suffix(R,j)) => EXISTS k : k<j & p(suffix(R,k))
  NEXT(p)(R):bool = p(suffix(R,1))
END temporal_logic
```

Fig. 6. Definition of linear temporal logic in PVS

```
modular[Glo_state,Loc_state: TYPE, smap:[Glo_state->Loc_state],
        Glo_event,Loc_event: TYPE, emap:[Loc_event->Glo_event]]
  : THEORY
BEGIN
IMPORTING temporal_logic[Glo_state,Glo_event]
  gs1,gs2: VAR Glo_state  ;  ge: VAR Glo_event  ;  le: VAR Loc_event
  lsp: VAR [Loc_state->bool]           % a local state predicate
  lep: VAR [Loc_event->bool]           % a local event predicate
  ltp: VAR [Loc_state,Loc_state->bool] % a local transition predicate

  HOLDS(lsp):TL = Holds(lsp o smap)
  TRANSITION(ltp):TL
    = Transition(LAMBDA (gs1,gs2): ltp(smap(gs1),smap(gs2)))
  OCCURS(lep):TL
    = Occurs(LAMBDA ge: EXISTS le: ge = emap(le) & lep(le))
END modular
```

Fig. 7. Modularisation of Temporal logic

We assume a state mapping from the global states of the system to the local states of the component; and an event mapping from local events to global events.

In Figure 7 we present a PVS theory that gives temporal logic a modular interpretation. The global and local states and events, and the state and event mappings are parameters of this theory. All formulae are to be interpreted in the world of global states and events, so we import `temporal_logic` (Figure 6) with parameters `Glo_State` and `Glo_Event`.

The theory defines local versions of the atomic formulae, expressing that local state predicates, local event predicates, or local transition predicates are true. The predicates in the local domain are mapped to global predicates, using the state and event mappings. The rationale is that the actual specification can use predicates on the locally known states and events.

For states this is quite easy: the local state predicate (`lsp`) can be composed with the state map (`smap`), yielding a global state predicate. For state transitions the situation is similar. For events the situation is different, because the event mapping is going in the other direction. A local event predicate (`lep`) occurs, if one of its elements corresponds via the event map (`emap`) to the global event that currently occurs.

3.4 Patterns in Requirements Specifications

The atomic formulae `HOLDS`, `TRANSITION` and `OCCURS`, together with the temporal operators `ALWAYS`, `BEFORE`, `NEXT` and the boolean connectives suffice to express many requirements. However, it appears that certain patterns reoccur very often. Abbreviations of such patterns are defined below. These abbreviations enable the engineer to express the requirements in a readable way.

Given a state predicate `lsp`, we say that it `BECOMES` true, if it becomes true in the current transition. State predicate `lsp` is a `PRECONDITION` of local event predicate `lep`, if `lep` only occurs in situations where `lsp` is true. The event predicate `lep` has the `EFFECT` described by the transition predicate `ltp`, if whenever an event that satisfies `lep` occurs, `ltp` holds. This yields the following definitions in PVS:

```
BECOMES(lsp):TL          = NEXT(HOLDS(lsp)) & NOT HOLDS(lsp)
PRECONDITION(lep,lsp):TL = ALWAYS (OCCURS(lep) => HOLDS(lsp))
EFFECT(lep,ltp):TL       = ALWAYS (OCCURS(lep) => TRANSITION(ltp))
```

Finally, we introduce some handy notation. In order to allow formulae like `OCCURS(x? OR y?)` and `HOLDS(pressed? => active?)`, we lift the boolean operators to operators on local state and event predicates, just by importing from Figure 5, `bool2pred[Loc_state]` and `bool2pred[Loc_event]`.

We also define an automatic conversion from events to event predicates. The advantage is that we can now write e.g. `OCCURS(raise)`, instead of the elaborate `OCCURS(LAMBDA le:le=raise)`. In PVS the conversion is defined as follows:

```
le1, le2: VAR Loc_event
event2pred(le1)(le2):bool = le1=le2
CONVERSION event2pred
```

In order to specify that the subcomponent `c` of the state doesn't change, we introduce the abbreviation `constant(c)` for the transition predicate `LAMBDA s1,s2: c(s1)=c(s2)`. In order to specify that e.g. the toggle event may not change the database-contents, we now write: `EFFECT(raise,constant(db))`. Note that the conversion `event2pred` also has to be applied.

4 Formal Specification of the Running Example

We now show how to express the requirements of the running example (Section 2) in PVS, using the framework developed in Section 3. Each component gives rise to three PVS theories: One to capture the data view, one to capture the view on behaviour, and an auxiliary one defining the set of events. The data view yields a type denoting the set of states that satisfy the invariants. The view on behaviour yields the requirements on runs, as a temporal logic formula.

4.1 States and events.

Figure 8 shows the state specification of the counter and database components in PVS. The states of the other components are defined analogously. See also Figures 1 and 2. The state of a component is based on a record, with fields for the state variables of the component (`value`, `z`), and for the states of its subcomponents (`x`, `y`). The invariants are modelled as predicates on this record. The type checker deals with `database_inv` as follows: naturals are a subtype of integers, and subtraction is closed under the integers, so the equality is on integers. Note that the invariant implies that `value(x(d)) >=`

```
counter_state: THEORY
BEGIN
 counter_state: TYPE = [# value: nat #]
END counter_state

database_state: THEORY
BEGIN
IMPORTING counter_state
  state_vars: TYPE = [# x,y: counter_state, z : nat #]
  database_inv(d:state_vars):bool =
     ( z(d) = value(x(d)) - value(y(d)) )
  database_state: TYPE = { d: state_vars | database_inv(d) }
END database_state
```

Fig. 8. Data view on counter and database in PVS

value(y(d)). The state is defined as the subtype consisting of those records that satisfy the invariant. The advantage is that states not satisfying the invariant are exposed by the type checker.

Also, for each component we need a type for the events. The disjoint union can be formalised in PVS by the DATATYPE construct. Figure 9 shows the event definition of the database and system components. The constructors of this datatype are the local events of the component (toggle), and the injections of the events of subcomponents (db, x). Note that we use overloading: depending on the context, x can denote the accessor in the states of the database, or the constructor in the events of the database. System events are for instance toggle, warn(raise) and db(y(increment)). Recognisers are also defined, such as db? of type [system_event->bool]. Destructors ev can also be used in the specification. The events of the other components are specified analogously (see Figures 1 and 3).

```
database_event: DATATYPE
BEGIN
IMPORTING counter_event
  x(ev:counter_event):x?
  y(ev:counter_event):y?
END database_event

system_event: DATATYPE
BEGIN
IMPORTING database_event, warning_event
  toggle: toggle?
  db(ev:database_event): db?
  warn(ev:warning_event): warn?
END system_event
```

Fig. 9. Event specification of system in PVS

4.2 Requirements on Behaviour.

We now formalise the requirements on the behaviour of the various components in detail. For the counter and database, we give complete specifications, to illustrate how the theories relate. For the other components we just give the temporal requirements, without showing the surrounding theory.

Counter component. Recall that we distinguish the global states and events of the combined system, from the local states and events of a component. For the specification of a component, the global states and events are unknown. Therefore, we put them as parameters to the theory on behaviour. Also, the state and event mappings are given as a parameter to this theory. Figure 10 presents the theory counter_behaviour. In order to list the parameters we just mentioned, the state and event definitions for the counter have to be

```
counter_behaviour
 [(IMPORTING counter_state, counter_event)
  State: TYPE, smap:[State->counter_state],
  Event: TYPE, emap:[counter_event->Event]
 ]: THEORY
BEGIN
IMPORTING modular[State,counter_state,smap,Event,counter_event,emap]

  increment_transition(c1,c2:counter_state):bool =
    ( value(c2) = value(c1) + 1 )
  requirements:TL = EFFECT(increment,increment_transition)
END counter_behaviour
```

Fig. 10. View on the behaviour of the counter

imported. As we want to use modular temporal logic, that theory is imported with the obvious actual parameters.

We can now specify the requirements in temporal logic. The only requirement on the counters is that the event `increment` shall have the `EFFECT` described by the transition predicate `increment_transition`. In PVS this is formalised straightforwardly.

Database component. The database component has no requirements on its own behaviour. So its requirements are merely the conjunction of the requirements of its subcomponents. In addition we must specify that there is no interference between the subcomponents, i.e. the occurrence of an event in one subcomponent doesn't change the state of any of the other subcomponents. These requirements can be found at the bottom of Figure 11.

In order to conjoin the requirements of the two subcomponents, we have to import the views on their behaviour, both defined in the same theory. They can be distinguished, because they are imported with different actual parameters. In order to distinguish them later on, we give the two imported theories a name, by the `..:THEORY = ..` mechanism of PVS, which also imports the theories.

Consider the subcomponent `x` of the database. In order to refer to the state of the database from the global state, we have to use `smap`. The projection (record accessor) `x` then yields the state of counter `x` within the database, so the complete state mapping is `x o smap`. Similarly, an event of counter `x` is referred to by the database component, using the injection (constructor) `x`, and then mapping the result to a global event by `emap`. So the complete event map is `emap o x`.

Note that the specification is independent of the actual state variables, events or requirements of the counters. It is also independent of the actual location of the database component in the system. This is required for a modular specification approach.

```
database_behaviour
  [(IMPORTING database_state, database_event)
   State: TYPE, smap:[State->database_state],
   Event: TYPE, emap:[database_event->Event]
  ]: THEORY
BEGIN
IMPORTING modular[State,database_state,smap,Event,database_event,emap]
x: THEORY = counter_behaviour[State,x o smap, Event, emap o x]
y: THEORY = counter_behaviour[State,y o smap, Event, emap o y]

  requirements:TL =
       x.requirements
    &  y.requirements
    &  EFFECT(x?,constant(y))
    &  EFFECT(y?,constant(x))
END database_behaviour
```

Fig. 11. View on the behaviour of the database

Warning component. In Section 2, we stated that two `raise` events shall be interleaved with a `close` or `withdraw` event. Similarly, two `close` or `withdraw` events shall be interleaved by a `raise` event. Finally, the initial step shall be a `raise` event. This can be specified in PVS as follows:

```
requirements: TL =
  ALWAYS(OCCURS(raise) =>
   NEXT(BEFORE(OCCURS(withdraw) OR OCCURS(close), OCCURS(raise))))
& ALWAYS(OCCURS(withdraw) OR OCCURS(close) =>
   NEXT(BEFORE(OCCURS(raise), OCCURS(withdraw) OR OCCURS(close))))
& BEFORE(OCCURS(raise),OCCURS(withdraw) OR OCCURS(close))
```

System component. Finally, the requirements on the system component define the requirements on the global system's behaviour. First, we define a transition to toggle the activation of the system, and we define what is regarded as a dangerous situation:

```
  toggle_transition(s1,s2:system_state):bool =
    ( s2 = s1 WITH [active? := NOT active?(s1)] )
  danger?(s:system_state):bool = z(db(s))>10
```

The requirements are specified in Figure 12. First, the effect of a `toggle` event and the initial value of `active?` is defined. Then the coordination between the warning component and the database is specified. A warning shall be raised only if `active?` holds, and it shall only be withdrawn when there is no `danger?`. This only forbids that warnings are raised or withdrawn at certain moments. In the next requirement, we state that whenever the situation becomes dangerous, a warning shall be raised eventually. Finally, we incorporate the standard requirements, viz. the requirements of the subcomponents shall hold, and the behaviour of subcomponents shall not interfere.

```
requirements:TL =
   EFFECT(toggle,toggle_transition)
&  HOLDS(NOT active?)

&  PRECONDITION(warn(raise),active?)
&  PRECONDITION(warn(withdraw),NOT danger?)
&  ALWAYS(BECOMES(danger?) => EVENTUALLY(OCCURS(warn(raise))))

&  db.requirements
&  warn.requirements
&  EFFECT(warn?,constant(db))
&  EFFECT(warn?,constant(active?))
&  EFFECT(db?,constant(active?))
```

Fig. 12. Behaviour specification of the system

5 Analysis of the Specification

The reader may have found certain problems in the formal specification of Section 4. It is the goal of the analysis to find such problems. Formal analysis consists of proving theorems on the specification. These proofs are carried out in PVS. Because the specification only contains definitions, and no axioms, this reasoning is sound even in the case that the specification would be inconsistent. We will discuss two different checks in the sequel. We also discuss the degree of automation in the proofs of the theorems in PVS.

5.1 Compatibility of State Transitions and Invariants

The first check is taken from [14]. The specification contains invariants, and also a number of state transitions. The intended meaning is that the design should meet the requirements on the state transitions as well as the invariants. In particular, if the counter values are incremented, z must be recomputed due to the integrity constraint $z = x.value - y.value$. The conjectures in PVS below check whether this is possible. They are derived from occurrences of `EFFECT` in the specification. Recall that states are subtypes, containing information on the invariants.

```
ds1,ds2: VAR database_state ; ss1,ss2: VAR system_state
CONJECTURE  FORALL ds1: EXISTS ds2: increment_transition(x(ds1),x(ds2))
CONJECTURE  FORALL ds1: EXISTS ds2: increment_transition(y(ds1),y(ds2))
CONJECTURE  FORALL ss1: EXISTS ss2: toggle_transition(ss1,ss2)
```

The first and third can be proved straightforwardly. The second proof fails. For `z(ds2)` we can only choose `z(ds1)-1`. The type checker then comes with the following subgoal (the assumptions are above the horizontal line, the proof obligations are below it):

```
[-1]    z(ds1) >= 0
  |-------
{1}    z(ds1) - 1 >= 0
```

We found a problem in the original specification! If $z(ds1) = 0$ and the value of y is incremented, then keeping the invariant $z = x.value - y.value$ would make z negative, which is forbidden because z is a natural number. The specification can be repaired, in several ways. Consulting domain experts might reveal that actually the distance between x and y is important, so that the corrected invariant reads $z = abs(x.value - y.value)$.

5.2 Scenarios satisfying the requirements

Is is also possible to validate the specification against a number of desirable and undesired scenarios. Given a scenario, it can be formally proved whether the specification correctly admits or refutes it. Note that if some scenario satisfies all requirements, this is a proof that the various requirements are consistent, in the sense that falsum is not derivable from their conjunction. Other contradictions in the specification can be found in this way too (see scenario R2 below). However, as this validation depends on the scenarios considered, it is always incomplete.

To start this verification, we first "close" the system, in the sense that we identify the global states and events with the states and events of the system specified in the previous section. The state and event mappings are instantiated with the identity function. Note that also subcomponents can be analysed in isolation by closing them.

```
IMPORTING system_state, system_event
IMPORTING system_behaviour[system_state,id,system_event,id]
```

Next we define the function `make_world` which, given boolean b and natural numbers i and j and a system event e, yields a state-event-record for the system, in which `active?` is set to `b`, the values of `x` and `y` are set to i and j, respectively, `z` is set to $i - j$ and the event is `e`. We constrain j such that $j \leq i$.

```
ws: warning_state
make_world(b,i,(j|i>=j),e):[#state:system_state,event:system_event#] =
(# state := (# active? := b,
     db := (# x:= (# value := i #), y:= (# value := j #), z:= i-j #),
     warn := ws #),
   event := e #)
```

Using this function, we can define the infinite run `R1`. It starts with `z` having the critical value 10. Then the events `toggle`, `x.increment` and `raise` happen, followed by an infinite number of `toggle`-events. (`add` is a predefined function on sequences.)

```
R1: run = add(make_world(FALSE,10,0,toggle),
           add(make_world(TRUE,10,0,db(x(increment))),
            add(make_world(TRUE,11,0,warn(raise)),
             LAMBDA i: make_world(even?(i),11,0,toggle))))
CONJECTURE system_behaviour.requirements(R1)
```

The conjecture is that this run satisfies all requirements for the system, defined in theory `system_behaviour`. The theorem is proved in PVS straightforwardly (see Section 5.3). This shows that this scenario is allowed by the specification. Hence the various requirements are not contradictory.

In the next scenario, counter `x` is repeatedly incremented, so on the i-th position in the run its value is i. The warning component is switched off by setting `active?` to false, so it is not necessary to raise warnings.

```
R2:run = LAMBDA i: make_world(FALSE,i,0,db(x(increment)))
CONJECTURE system_behaviour.requirements(R2)
```

Somewhat surprisingly, this conjecture proved to be false. Directed by the resulting unprovable subgoal, the specification was inspected, revealing another mistake. After 10 increments, `danger?` becomes true, and a warning must be raised eventually; this however should not occur, because the warning component is off. This error can be repaired in several ways. The most natural is to weaken one of the requirements to

```
ALWAYS(BECOMES(danger?) & active? => EVENTUALLY(OCCURS(warn(raise))))
```

After some experiments, we found the following R3, which is a "bad" scenario. Its event trace is $toggle.raise.toggle^{\omega}$. This is not an intended run of the system, because a warning is raised without reason.

```
R3:run =  add(make_world(FALSE,0,0,toggle),
            add(make_world(TRUE,0,0,warn(raise)),
             LAMBDA i: make_world(even?(i),0,0,toggle)))
CONJECTURE system_behaviour.requirements(R3)
```

R3 was shown by PVS to satisfy all requirements. This indicates that the specification is not complete. Warnings should be raised if and only if `active?` is on, and the situation is dangerous. We forgot to specify the "only if" part.

5.3 Degree of Automation

The theorems are proved semi-automatically in PVS. The `GRIND`-command performs unfolding of the definitions, basic arithmetic and skolemisation of quantifiers. This does the bulk of the work, eliminating all references to temporal logic, and checking basic arithmetic facts. This automation not only saves a lot of work, but it also makes the proofs more robust. After small changes in the specification, most proofs can simply be rerun.

Three kinds of subgoals had to be discharged by hand. Theorems about odd and even numbers (for instance, the successor of an odd number is even). Such theorems should typically be included in the standard library. Secondly, the system didn't always automatically infer inequalities, like `warn(raise)`$\neq$ `warn(close)`. This seems to be an omission in the decision strategy of PVS.

Finally, proving `EVENTUALLY(p)` or `BEFORE(p,q)` boils down to finding a point in the run where `p` is true. The system cannot instantiate this existential quantifier automatically. This is unavoidable, because the logic is undecidable, due to its expressivity.

6 Conclusion

Results. In this paper, we have described a modular approach to the formal specification of the requirements on embedded systems. In this approach a specification consists of two views, defining the data and the behaviour of a system. Our approach allows abstract specifications, because the requirements are stated in an assertional way. The requirements on data are specified by invariants on states of the system. Behaviour is specified by temporal logic formulae on runs of the system.

The proposed method is modular, because components can be specified and analysed in isolation, and the views of several components can be combined in an easy way. Combining states corresponds to taking the Cartesian product. Events can be combined by taking disjoint union. The requirements on behaviour can be combined by simply taking the conjunction.

A mathematical framework supporting this approach has been developed. We have implemented this framework in the specification language and theorem prover of PVS. This gives rise to a non-trivial application of PVS. The implementation hinges on the particular feature of PVS, that theories can be parameterised by types and terms. Each component in the specification leads to a number of theories, parameterised by certain mappings between the components. The generic part of the implementation can be reused for other specifications without change. The method was illustrated by formalising the requirements of a miniature embedded system. This specification could then be analysed, revealing some errors in the specification. The analysis concentrated on compatibility of invariants and state transitions and on validating and rejecting scenarios.

Related Work. We briefly summarise some existing specification formalisms. The scheme calculus of Z [17], makes specifications modular with respect to the data view. Z specifications incorporate state invariants and transitions but no events. From a Z specification the interaction with the environment cannot be unambiguously inferred (this is also true of VDM [6]). This observation is not new: in Object-Z [15] an order on actions can be specified with temporal logic formulae; see also [7], where deontic logic is used to describe when certain actions are permitted or obliged.

Process algebras, on the other extreme, are completely control-oriented. The state is only implicitly present, as a kind of “program counter”. Formalisms like μCRL [4], and I/O-automata [9] have also a data description, but they remain control-oriented; only the visible actions of processes count. Moreover, specifications in these formalisms tend to be too operational for requirements engineering. Finally, formalisms based on temporal logic [10, 1], and TLA [8], have a notion of state and state transitions, and behaviour can be specified by temporal assertions, but these formalisms lack a principle of hierarchically structured states, so they are not modular with respect to the data view.

In graphical languages, like UML [3] and STATEMATE [5], the emphasis is on expressibility and scalability, rather than on the underlying mathematical framework.

Comparing our approach with object-oriented analysis [2], we see some similarities and differences. As in object orientation, we can specify components in isolation, and also have aggregates of components. These aggregates can form a hierarchy. Also multiple occurrences of the same component are allowed. However, our approach doesn't include object creation. We also deliberately don't have a mechanism of message passing or encapsulation of data or internal events. We think that these mechanisms easily lead to implementation biased specifications, and are not helpful for requirements engineering.

Our scenario-based analysis resembles model checking. However, a model checking tool cannot be effectively deployed, as long as there is no (finite) operational description of the system. Model checking could be useful during the design of the system, where operational descriptions emerge.

Future work. We have chosen a very simple model, viz. sets of runs. Extensions with real-time will be considered, and also ready/failure trace semantics, in order to express that certain input events should be enabled (cf. [11]). The analysis can then be extended by proving other interesting properties of a specification, for instance robustness, meaning that the specified system accepts all possible input.

Research on the smooth integration of requirements specification and design is needed. In design, data and behaviour of components are refined, and structural components are combined by taking parallel composition. To allow refinement and composition, the model has to be extended, with for instance stuttering steps [8], communication primitives and encapsulation. The work on compositionality, e.g. [1] may provide useful insights.

Finally, this paper is not concerned with the design of a concrete specification language, which is convenient and readable for engineers. Such a language might include restricted versions of class diagrams, finite automata, tabular representations or process expressions. A user interface might hide the administration of parameters, mappings and importings, which can be generated automatically from Figure 1. Partial specifications must be mapped systematically into the common underlying mathematical framework, in order to obtain a sound multi-paradigm specification method (cf. [18]).

Acknowledgements. We like to thank Roel Bloo and Dieter Hammer from the Eindhoven University of Technology for critical comments on previous versions of this paper. We are also grateful to Martin Streng from Hollandse Signaalapparaten B.V. for discussions on the applicability of our method. The referees suggested valuable improvements to the paper.

References

1. H. Barringer, R. Kuiper, and A. Pnueli. Now you may compose temporal logic specifications. In *ACM Symposium on Theory of Computing (STOC '84)*, pages 51–63. ACM Press, 1984.
2. G. Booch. *Object Oriented Analysis and Design with Applications.* The Benjamin Cummings Publishing Co. Inc., 1991.
3. M. Fowler and K. Scott. *UML Distilled: Applying the Standard Modeling Object Language.* Object Technology Series. Addison-Wesley, 1997.
4. J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Algebra of Communicating Processes, Utrecht 1994*, Workshops in Computing, pages 26–62. Springer, 1994.
5. D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and A. Shtul-Trauring. STATEMATE; a working environment for the development of complex reactive systems. In *Proc. of the 10th Int. Conf. on Software Engineering*, pages 396–406, Singapore, 1988. IEEE Computer Society Press.
6. C.B. Jones. *Systematic Software Development using VDM.* International Series in Computer Science. Prentice-Hall, Inc., 2nd edition, 1990.
7. S. Khosla and T.S.E. Maibaum. The prescription and description of state based systems. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Temporal Logic in Specification*, volume 398 of *LNCS*, pages 243–294. Springer, 1987.
8. L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.
9. N.A. Lynch. I/O automata: A model for discrete event systems. In *Proc. of 22nd Conf. on Inf. Sciences and Systems*, pages 29–38, Princeton, NJ, USA, 1988.
10. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems – Specification.* Springer, 1992.
11. E.R. Olderog and C.A.R. Hoare. Specification-oriented semantics for communicating processes. *Acta Informatica*, 23(1):9–66, 1986.
12. S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Proc. of the 8th Int. Conf. on Computer Aided Verification*, volume 1102 of *LNCS*, pages 411–414. Springer, 1996.
13. S. Owre, N. Shankar, J.M. Rushby, and D.W.J. Stringer-Calvert. *PVS Language Reference.* Computer Science Laboratory, SRI International, Menlo Park, CA, September 1998.
14. J.C. van de Pol, J.J.M. Hooman, and E. de Jong. Formal requirements specification for command and control systems. In *Proc. of the Conf. on Engineering of Computer Based Systems*, pages 37–44, Jerusalem, 1998. IEEE.
15. G. Rose. Object-Z. In S. Stepney, R. Barden, and D. Cooper, editors, *Object Orientation in Z*, Workshops in Computing, pages 59–77. Springer, 1992.
16. J. Rushby. Formal methods and the certification of critical systems. Technical Report SRI-CSL-93-7, SRI International, Menlo Park, CA, 1993.
17. J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 2nd edition, 1992.
18. P. Zave and M. Jackson. Where do operations come from? A multiparadigm specification technique. *IEEE Trans. on SE*, 22(7):508–528, 1996.