Refinement in Requirements Specification and Analysis: a Case Study

Edwin de Jong
Hollandse Signaalapparaten
P.O. Box 42
7550 GD Hengelo
The Netherlands
edejong@signaal.nl

Jaco van de Pol
CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands
Jaco.van.de.Pol@cwi.nl

Jozef Hooman
University of Nijmegen
P.O. Box 9010
6500 GL Nijmegen
The Netherlands
hooman@cs.kun.nl

Abstract

This paper presents a formal method for requirements specification and analysis. Using this method some techniques for step-wise refinement are studied. During the early phases of system development, where the exact requirements are yet unclear, these techniques allow to write incomplete and global specifications, which during successive steps can be refined and completed. At each step the method supports formal analysis of the specification. In particular two abstraction techniques are studied: nondeterminism and uninterpreted symbols. These techniques are explored using a realistic case study, that was inspired by the specification of an existing naval command and control system. Specifications are written and analysed using the language and proof checker of PVS.

1. Introduction

The strength of formal system specifications is that they are written with mathematical precision. Using a formal specification language ensures that the system specification is unambiguous. A formal semantics provides for a model of the system, that can be analysed with mathematical rigour. Powerful tools, like theorem provers, that are nowadays available, make the analysis tractable.

In the early phases of system development, where most of the requirements are yet unclear, this strength however usually proves to be a weakness. At this stage of development, any method that enforces completeness of the specification is not considered very helpful. At the same time, however, particularly during the system requirements specification phase, there is a clear need of methods, techniques, and tools that help in assessing the correctness and quality of the specification. Errors and omissions in the requirements specification tend to propagate through all later phases, until they are detected. It is well known, that errors that are made early and detected late, are relatively expensive to repair.

In order to support requirements specification and analysis, a formal method should be able to cope with incomplete, in the sense of "yet unfinished", and very global descriptions. During successive stages, it should be possible to complete and refine the initial specification, until the intended system is precisely described. At each stage the intermediate specification should be suitable for analysis; in fact, the outcome of this analysis should provide a guiding principle for the decisions in the next iteration.

In this paper we present a formal method for requirements specification and analysis. Using this method we experiment with some techniques for step-wise refinement by means of a realistic case study. The method is an extension of earlier work [2] and is based on a specification in higher-order logic, that defines a state-transition system which accepts input events and emits output events. Using this framework, we study in this paper two abstraction techniques that support step-wise refinement: nondeterminism and parameterized specifications. Using nondeterminism, the system engineer at first globally delineates the intended system behaviour. Initially the specification may implicitly contain some unintended behaviours as well. During successive refinement steps, the amount of nondeterminism can be gradually reduced until precisely the intended system behaviours remain. Formally, each refinement step carries a proof obligation, since it should be verified that the refined specification does not introduce any behaviour that is excluded by the higher-level specification.

Using parameterized specifications, the system engineer is able to write incomplete descriptions of the system, where the parameters refer to (yet) uninterpreted parts of the specification. These descriptions can later on be refined by instantiating the parameters with actual definitions. In the parameterized specification usually various assumptions are stated about the parameters. It is a proof obligation of each refinement step to formally verify that these assumptions are met by the actual parameters.

We have implemented our method using the language and interactive proof checker of PVS [9]. We regard integrated tool support as an indispensable means for any practical application of formal requirements specification and analysis. In our method the proof checker of PVS is used in performing various types of analyses of the specification, and in particular to verify the correctness of each refinement step.

This paper is organized as follows. In Section 2 we first outline the general specification and analysis method, including its tool support. To illustrate, we discuss in Section 3 a case study that was inspired by the specification of an existing naval command and control system. The specification starts from a concise and global description of the system, which we refine and complete in Section 4 by two successive refinement steps. The first step focuses on behavioural requirements by reduction of nondeterminism. The second step fills in some details that were left open in the initial specification by instantiating some of the specification's parameters with actual definitions. We analyse both refinement steps and prove that they extend the requirements from the higher-level specification. In Section 5 we conclude with an evaluation of the method, a brief discussion on related work, and some indications for future research.

2. Specification and Analysis Method

2.1. Specification

Ideally a requirements specification should consider the system as a black box, defining the system's output given a series of input events. Particularly for large-scale systems, however, it is not feasible in practice to define the relationship between input and output events directly. For this reason, an internal system state is introduced as an auxiliary means to specify this relationship. On the basis of input events, the system accumulates information on the current state of affairs in its environment. This information is represented by the system's internal state. The internal state, in turn, is used to decide which output events occur.

We make the following modeling assumptions: Input events are atomic, so they occur instantaneously, one at a time. Each input event causes a state transition, which also takes place instantaneously. A state transition may trigger (cause) one or more output events.

To formalise these ideas, we propose the following system model, that defines the basic building blocks of a formal specification.

A. Static system interface. The static interface between the system and its environment is defined by the kind of input and output events that are handled by the system. Since we want events to carry arbitrary parameters, the sets of possible input and output events may be infinite. So formally the static system interface is determined by:

- a set of input events I;
- a set of output events O.

B. Information model. As mentioned earlier, the internal state of the system is introduced as an auxiliary means to define the relationship between input and output events. The information model determines the internal state space of the system; it is defined by:

- a list of state variables and their types - together these span the system's state space S;
- a set of invariant conditions $Inv \subseteq S$;
- a set of allowed initial states $S_0 \subseteq S$.

C. Behavioural Model. The system's behaviour is defined as a relationship between input and output events. This relationship is specified indirectly by means of state transitions. For each input event it is defined which state transition is caused by it, and conversely, for each output event it is specified by which state transition it is triggered. Formally we define:

- a state transition function $\Delta : I \to \wp(S \times S)$, and
- an output trigger function $\Lambda : O \to \wp(S \times S)$.

Here $\wp(S \times S)$ denotes the powerset of $S \times S$.

D. Environment. Finally, any assumptions about the system's environment are explicitly specified. These assumptions are formulated as preconditions on the input events. These preconditions can be seen as requirements that the system's environment should meet: as long as these conditions are met, the system shall exhibit well-defined behaviour. In this way, the integration of the system into a certain environment can be verified to work. Formally these preconditions are defined as:

- a function $Pre \in I \to \wp(S)$, that maps each input event onto a condition on the system's internal state.

A system specification, constructed from the above building blocks, is to be interpreted as follows. Initially, the system is in some state $s \in S_0 \cap Inv$. Assume that during execution, the system is in some state $s \in S$, and that input event $i \in I$ occurs. Provided that $s \in Pre(i)$, the system then moves (nondeterministically) to some state t, such that $(s, t) \in \Delta(i)$ and $t \in Inv$ (if such a t exists). During this transition, all output events o, for which $(s, t) \in \Lambda(o)$ are triggered. If such a t cannot be found or if $s \notin Pre(i)$, the system is blocked (deadlock), which is an undesirable situation.

Note that, using this interpretation, we need not specify how the invariant is maintained by the various state transitions. Instead, the next state t is taken from the intersection of $\Delta(i)$ and Inv. In this way, the specification tends to become less operational and more declarative.

2.2. Analysis

For the analysis of a requirements specification, a distinction can be made between verification and validation. By *verification*, the intrinsic quality of the specification is addressed. Special attention is given to the verification that the various blocks of a specification fit together consistently. It cannot be verified however, whether the intended system has been specified. This check is the purpose of *validation*. So validation addresses the question whether the correct system is specified, while verification addresses the question whether a system is specified correctly. Though formal methods can be a valuable tool in validating a requirements specification, by challenging the specification by putative theorems, we focus in this paper on verification.

Parsing and type checking form the first sanity check of a specification. Together, these checks reveal a lot of errors. In a strongly typed language, most typos are detected by parsing and type checking, and also a number of conceptual problems are detected by the latter. In a strongly-typed logical language, type checking can also guarantee semantical completeness, for instance totality of functions, exhaustiveness of case distinctions and tables etc.

Besides parsing and type checking, it is important for the proposed specification model to verify that the set of possible initial states is non-empty. Recall that all states have to satisfy the invariant, including the initial state. Note that this establishes that the invariant is consistent, since we found a model.

Proof obligation 1. $\exists s.\, s \in S_0 \cap Inv$

Another proof obligation is to establish deadlock freedom in any environment that guarantees the precondition. Assuming that some input event i occurs in state s, the system is blocked either if s does not satisfy $Pre(i)$ or if all candidate next states t (i.e. $(s, t) \in \Delta(i)$) do not meet the invariant condition. By establishing deadlock freedom, we also show that the state transitions and the invariants are consistent (under assumption of the precondition), because the existence of transitions that satisfy both is proved. This proof obligation can be formally stated as follows. Note that we can safely assume that s satisfies the invariant.

Proof obligation 2.
$\forall i \in I. \forall s \in Inv \cap Pre(i). \exists t \in Inv. (s, t) \in \Delta(i)$

Other general proof obligations for specifications can be stipulated here. Especially, we do not yet have an obligation to check that the specification of the output events is reasonable. As in [5], it could be required that the state transitions are deterministic. In general, however, we do not require this of a specification, since we will exploit nondeterminism as an abstraction technique to support step-wise refinement.

2.3. Tool Support

In order to have tool support at this experimental stage of research, we use an existing general-purpose theorem prover. We have implemented the proposed specification and analysis method in PVS (Prototype Verification System) [7, 9]. PVS is a specification language integrated with support tools and a theorem prover, developed at SRI, Stanford.

PVS supports the proposed specification method in various ways. Firstly, specifications are written in the language of PVS, which is based on strongly-typed higher-order logic. The specification can be distributed over a number of theories, which are parameterizable, and can import each other. Each theory consists of a number of definitions and theorems. These can be made available in other theories, by the importing-mechanism. A theory can also make assumptions on its parameters.

Secondly, using PVS a specification can be parsed automatically, and type checked. Type checking is to a large extent automatic, but due to the rich type system certain checks are undecidable in general. These cases lead to extra type check conditions (TCCs). A theory can only be regarded type correct, if these TCCs are proved.

Finally, the TCCs, together with the proof obligations and the putative theorems declared in the specification, can be proved using the theorem prover. This is an interactive tool, because theorem proving in higher-order logic is undecidable. So the user interactively provides commands to apply the proof rules. However, PVS supports a lot of automation to decrease the number of user interactions. We like to mention here: term rewriting techniques, decision procedures for linear arithmetic, model checking on finite state systems, a BDD-based decision procedure for propositional logic, and advanced heuristics to deal with quantifiers.

Due to space limitations, we are not able to explain all details of PVS. The intentions of the specifications that we present in this paper, however, should be clear from the accompanying text. For more details about PVS, we refer to e.g. [8].

3. Case Study: Initial Specification

To illustrate and evaluate the proposed method, we formally specify and analyze the requirements of a subsystem that were inspired by a naval command and control system. The subsystem that we consider here is Track Management (TM). First we informally describe the requirements of TM, focusing on automatic track fusion, which is one of the main tasks of TM. The initial description is still quite global; many details are missing and some requirements are missing as well.

We will apply the method from Section 2 using PVS to formally specify TM. The specification is decomposed into blocks A through D as previously discussed. Since the initial description of TM is still very general, we will exploit nondeterminism in the state transition relation Δ to coarsely delineate the intended system behaviour. Additionally, we will make frequent use of uninterpreted terms for parts of the specification that are initially left open. In the next section the initial specification will be completed by two successive refinement steps.

3.1. Track Management

In command and control systems a track models a target that has been detected in the environment. A track consists of the target's measured position, velocity, identification, etc. Tracks occur on (at least) two levels:

- sensor tracks, as reported by the system's sensors;
- system tracks, as compiled and maintained by TM.

A target in the environment may be simultaneously detected by multiple sensors. Since the observations are subject to measurement noise, this may result in different sensor tracks. One of the tasks of TM is to create and maintain a single system track for one or more sensor tracks that originate from the same target. To this end, TM applies a set of predefined criteria to determine whether a sensor and system track correlate.

The sensors interact with TM by initiating new sensor tracks, and updating or wiping existing sensor tracks. TM maintains a track database, containing the current sets of sensor and system tracks, their relationship, and the derived system tracks' state. TM reports to the operator (and other subsystems) by initiating, updating, or wiping system tracks. In addition, TM shall generate a warning if a sensor track and its related system track decorrelate, alerting the operator of a 'split target' situation.

3.2. Static System Interface

The formal specification of TM starts from the system interface, which is defined in terms of the types of input and output events. From the description of TM we see that there are three types of input events `new`, `update` and `wipe`, with the purpose of initiating, updating and deleting sensor tracks. Each input event carries a sensor track as parameter. The `new` and `update` events additionally carry the sensor track's current state, reporting on its position, velocity identification, etc. Because at this stage it is not yet clear exactly which attributes the sensor track's state will be composed of, we use PVS uninterpreted type declarations, using the keyword `TYPE` or `TYPE+`; the latter indicates that the type is non-empty. To facilitate step-wise refinement, we put these uninterpreted terms into the parameter list of the PVS theory that represents the initial specification of TM. Later the initial specification can be refined by (partially) instantiating the parameters with actual definitions. Using abstract data types to define the set of input events `IEvents`, the PVS theory that specifies TM starts as follows.

```
initial_spec
 [ Sensor_track: TYPE,
   Sensor_track_state: TYPE+,
   ...
 ] : THEORY
BEGIN

IEvents: DATATYPE
BEGIN
  new(s: Sensor_track,
      x: Sensor_track_state):
      new_sensor_track?
  update(s:Sensor_track,
         x:Sensor_track_state):
         update_sensor_track?
  wipe(s: Sensor_track): wipe_sensor_track?
END IEvents

...

END initial_spec
```

The set of output events `OEvents` is defined similarly. We introduce three output events: `new`, `update`, and `wipe`, this time defined on system tracks. Again we use uninterpreted types `System_track:TYPE` and `System_track_state:TYPE+` to refrain from details and add these types to the parameter list of the theory. Additionally, the operator shall be warned of decorrelating sensor and system tracks, which we model by the output event `warn`.

```
OEvents: DATATYPE
BEGIN
  new(t: System_track,
      y: System_track_state):
      new_system_track?
  update(t: System_track,
         y: System_track_state):
         update_system_track?
  wipe(t: System_track): wipe_system_track?
  warn(s: Sensor_track): warn?
END OEvents
```

3.3. Information Model

The informal description of TM states that "TM maintains a track database, containing the current sets of sensor and system tracks, their relationship, and the derived system tracks' state." We model the system's internal state as a record consisting of these four components.

```
State: TYPE = [#
  sensor_tracks: setof[Sensor_track],
  system_tracks: setof[System_track],
  state: [System_track->System_track_state],
  related: pred[[Sensor_track,System_track]]
#]
```

From the description of TM it becomes clear that the relation between sensor and system tracks is restricted to those that are actually present in the current state. Furthermore, each system track should be related to at least one sensor track, and conversely, each sensor track should be related to precisely one system track. These requirements can be formulated as an invariant condition on the state. In this way, we fix some global properties of the system, without indicating how they should be maintained.

```
s: VAR Sensor_track
t: VAR System_track
X, Y: VAR State

constraint1(X):bool = FORALL s,t :
  related(X)(s,t) =>
  sensor_tracks(X)(s) & system_tracks(X)(t)
constraint2(X):bool = FORALL s:
  sensor_tracks(X)(s) =>
  exists1! t: related(X)(s,t)
constraint3(X):bool = FORALL t:
  system_tracks(X)(t) =>
  EXISTS s: related(X)(s,t)
Invariant(X):bool = constraint1(X)
  & constraint2(X) & constraint3(X)
```

We complete the information model by specifying the set of allowed initial states. In an initial state we require that the set of sensor tracks is empty.

```
initial(X):bool = empty?(sensor_tracks(X))
```

Note that the invariant restricts the set of system tracks and their relationship. In the analysis of the specification (Section 3.6) we will show that there exists at least one initial state satisfying the invariant.

3.4. Behavioural Model

The behavioural model defines the relationship between input and output events by mapping input events to state transitions, and similarly by mapping output events to state transitions. For TM we arrive at the following pair of tables, binding each input or output event to a (nondeterministic) state transition. The next step in the specification will be to define these state transitions.

```
ie: VAR IEvents
oe: VAR OEvents
x: VAR Sensor_track_state
y: VAR System_track_state

IEvent_mapping(ie): [State,State->bool] =
  CASES ie OF
    new(s,x)   : new_sensor_track(s,x),
    update(s,x): update_sensor_track(s,x),
    wipe(s)    : wipe_sensor_track(s)
  ENDCASES

OEvent_mapping(oe): [State,State->bool] =
  CASES oe OF
    new(t,y)   : new_system_track(t,y),
    update(t,y): update_system_track(t,y),
    wipe(t)    : wipe_system_track(t),
    warn(s)    : detect_decorrelation(s)
  ENDCASES
```

Due to space considerations we discuss only a representative part of the complete specification. In particular, we consider the state transitions `new_sensor_track` and `detect_decorrelation`.

When a new sensor track is reported, TM may relate the sensor track to an existing system track, provided that certain correlation criteria are met. At this stage of the specification, however, we are not yet concerned with the actual correlation test to be used. Hence, we introduce the correlation test as an uninterpreted relation `correlates(x,y):bool` between a sensor and a system track state, and add it to the parameter list of the theory.

For the state transition `new_sensor_track` we require that the new sensor track is added to the current set of sensor tracks. Note how the invariant condition on the state, particularly `constraint2`, already requires that the sensor track be related to *some* system track. Additionally, we now state that the sensor and system track shall satisfy the correlation test, if the system track already exists. Using the informal description of TM as a basis, we are at this stage of requirements specification not yet concerned with the exact conditions under which a new system track shall be created or an existing system track shall be updated by a new sensor track. Neither do we specify how the system track state shall be initialized or updated. Using nondeterminism in the definition of `new_sensor_track`, we leave all possible implementations yet open.

```
new_sensor_track(s,x)(X,Y): bool =
  sensor_tracks(Y) = add(s,sensor_tracks(X))
  & FORALL t: related(Y)(s,t)
            & system_tracks(X)(t)
            => correlates(x,state(X)(t))
```

The state transitions of the two remaining input events `update` and `wipe` can be defined similarly. Our next concern are the state transitions that trigger the output events. As an example we consider the state transition that produces a warning to the operator in case a sensor track and its related system track decorrelate. Formally stated, a decorrelation is detected if a sensor track `s` is related to different system tracks in subsequent states `X` and `Y`:

```
t1, t2: VAR System_track

detect_decorrelation(s)(X,Y): bool =
  EXISTS t1,t2: related(X)(s,t1)
                & related(Y)(s,t2) & t1/=t2
```

3.5. Environment

We complete the initial requirements specification by making the assumptions on the system interface explicit. Here the interface is restricted to the interaction between the system's sensors and TM. We expect the sensors to obey the 'track life-cycle'. This means that a sensor is assumed to initiate only fresh sensor tracks and, conversely, to update or wipe only existing sensor tracks. These assumptions are formulated as a precondition on the system's internal state:

```
Precondition(ie)(X): bool =
  CASES ie OF
    new(s,x)    : NOT sensor_tracks(X)(s),
    update(s,x) : sensor_tracks(X)(s),
    wipe(s)     : sensor_tracks(X)(s)
  ENDCASES
```

3.6. Analysis of the Initial Specification

We start the analysis of the initial specification by verifying the proof obligations that were introduced in Section 2.2. In PVS we have implemented a generic theory that automatically generates the proof obligations when it is imported into a specification [4].

When we import this theory into the specification of TM, PVS automatically generates the following two proof obligations while type checking.

```
IMPORTING1_TCC1: OBLIGATION
  EXISTS X: initial(X) & Invariant(X)

IMPORTING1_TCC2: OBLIGATION
  (FORALL (i: IEvents, X: State):
     Precondition(i)(X) & Invariant(X)
     => (EXISTS (Y: State):
          Invariant(Y)
          & IEvent_mapping(i)(X,Y)))
```

The first proof obligation is to check that there exists at least one initial state `X` that satisfies the invariant condition on the system's internal state. In the interactive prover, this property can be proved by first instantiating `X` to the state, where the sets of sensor and system tracks and their relationship are empty. Then the single PVS command `(GRIND)` finishes the proof by automatically verifying that all conditions on `X` are met. Note that this also proves consistency of the invariant condition, because we have found at least one state by which it is met.

The second proof obligation concerns the absence of deadlock in any environment that guarantees the precondition. Using the interactive proof checker of PVS, the proof proceeds by case distinction over the possible input events. For each input event an auxiliary lemma can be formulated, stating that a transition to a new state `Y` is possible, given that the precondition holds in the current state `X`. For the input event `new` the lemma reads as follows.

```
case_new: LEMMA
   Invariant(X) & Precondition(new(s,x))(X)
=> EXISTS Y: Invariant(Y)
          & new_sensor_track(s,x)(X,Y)
```

For the remaining two input events similar lemmas can be formulated. The proofs are based on the construction of a state `Y` that is shown to satisfy the invariant condition and the state transition relation. It appears that the proof of the above lemma requires the assumption that a fresh `System_track` can always be found in the construction of `Y`. In PVS this assumption on the type `System_track` can be explicitly formulated in the theory `initial_spec`.

4. Refinement Techniques

The formal specification of TM is still very global and is lacking many details. For instance, the actual sensor and system track states and the correlation criteria were left out from the initial specification. Furthermore, the specification still allows some unintended implementations; a potential implementation of `new_sensor_track`, for example, can easily decide to always create a new system track.

Here we discuss two ways of refining the initial specification, based on (1) reduction of nondeterminism in state transitions, and (2) instantiation of uninterpreted terms. We discuss both techniques by successively applying them to the initial specification of TM.

4.1. First refinement

Unintended implementations of the state transition `new_sensor_track` can be ruled out by extending the initial requirements as follows. If a new sensor track is received by TM, a new system track shall be initiated only if the correlation tests on all existing system tracks fail; existing system tracks shall remain unchanged. Otherwise the new sensor track is related to a correlating system track, and the correlating system track's state shall be updated accordingly. Again, all other system tracks shall remain unchanged.

To formally define these requirements, we introduce two auxiliary state transitions `new_system_track` and `correlating_system_track` to explicitly cover both cases. At this stage we are not yet concerned with the actual calculations on the system track state, so we add uninterpreted functions `initiate` and `update` to the parameter list of the specification.

```
first_refinement [
  Sensor_track, Sensor_track_state,
  System_track, System_track_state: TYPE+,
  correlates:[Sensor_track_state,
              System_track_state ->
              bool],
  initiate:[Sensor_track_state ->
            System_track_state],
  update:[Sensor_track_state,
          System_track_state ->
          System_track_state]
]: THEORY
BEGIN

...

END first_refinement
```

In this theory the state transitions `new_system_track` and `correlating_system_track` are defined as follows.

```
new_system_track(s,x)(X,Y):bool =
  (FORALL t: system_tracks(X)(t) =>
             NOT correlates(x,state(X)(t)))
  & EXISTS t:
    (NOT system_tracks(X)(t))
    & Y =
      (# sensor_tracks :=
           add(s,sensor_tracks(X)),
         system_tracks :=
           add(t,system_tracks(X)),
         related := add((s,t),related(X)),
         state := state(X)
           WITH [t := initiate(x)]
       #)

correlating_system_track(s,x)(X,Y):bool =
  EXISTS t:
    system_tracks(X)(t)
  & correlates(x,state(X)(t))
  & Y = X WITH
    [sensor_tracks :=
       add(s,sensor_tracks(X)),
     related := add((s,t),related(X)),
     state := state(X)
       WITH [t := update(x,state(X)(t))]]
```

Note that both state transitions are still nondeterministic. In `new_system_track` any fresh system track is allowed, and `correlating_system_track` does not specify which system track shall be chosen, if there is more than one correlating system track.

The `new_sensor_track` state transition from the initial specification can now be redefined as follows.

```
new_sensor_track(s,x)(X,Y):bool =
   new_system_track(s,x)(X,Y)
OR correlating_system_track(s,x)(X,Y)
```

The other state transitions from the initial specification for updating and wiping sensor tracks can be refined in a similar fashion.

We are obliged to verify that the refined state transitions do not introduce any behaviour that is excluded by the initial specification. Formally, in terms of the basic building blocks from Section 2.1, for any state transition $\Delta(e)$ that is refined by state transition $\Delta'(e)$, it should be proved that, for any $x, y \in Inv$, if $(x, y) \in \Delta'(e)$ and $x \in Pre(e)$, then $(x, y) \in \Delta(e)$. We have implemented a generic PVS theory that, when imported into the refined specification together with the initial specification, automatically generates the corresponding proof obligation in PVS. For the refined specification of TM, the proof can be established using the interactive proof checker of PVS by case distinction over the various state transition relations. In particular, for the refined transition `new_sensor_track` we can prove the following lemma.

```
L1: LEMMA
  new_sensor_track(s,x)(X,Y) & Invariant(X)
  & Invariant(Y) & Precondition(new(s,x))(X)
  => initial_spec.new_sensor_track(s,x)(X,Y)
```

Finally, we should verify the proof obligations from Section 2.2 for the refined specification. Since the refined specification has more requirements, that is, allows fewer state transitions, the proofs need to be established explicitly. The proofs, however, are very similar to those of the initial specification – in fact, constructing the next state is more straightforward, due to the more operational definition of the state transitions in the refined specification.

4.2. Second refinement

In the initial specification as well as in the previous refinement, some types and constants were left uninterpreted, thus allowing some degree of freedom in the implementation. In the second refinement of TM, we now actually assign a meaning to these uninterpreted symbols. To facilitate this refinement step, the uninterpreted symbols were specified in the parameter list of the previous specification. In this way, a set of specifications is obtained, one for each value of the parameters. A refinement of the specification can be seen as a (partial) instantiation of these parameters, thus decreasing the set of possible implementations.

The only proof obligation is that the refined specification is indeed an instantiation of the original specification. In particular, it must be checked that the actual parameters satisfy the assumptions put on the parameters. These checks are automatically generated by the PVS type checker.

To illustrate this, we continue with the requirements specification of TM with a second refinement step. It is decided that the sensor and system track states shall consist of three dimensional position and velocity vectors, and identification of the target. In addition, sensor tracks shall contain information regarding their source. A sensor and system track shall correlate if their mutual distance is within some predefined margin and their identification is not conflicting.

In a similar fashion the initiation and updating of system tracks can be described in more detail, resulting in a refined specification that can be outlined as follows. Note how the `IMPORTING` clause at the end of the PVS theory actually instantiates the specification of the first refinement with the definitions of the second refinement.

```
second_refinement
 [ Sensor_track, System_track,
   Source: TYPE+,
   Margin: {x:real | x>0}
 ]: THEORY
BEGIN

Identification: TYPE =
  {friend, hostile, pending}

Vector: TYPE = [# x,y,z: real #]

Sensor_track_state: TYPE =
[# source: Source,
   identification: Identification,
   position, velocity: Vector
#]

System_track_state: TYPE =
...

correlates(x,y):bool =
  distance(x,y) <= Margin &
  NOT conflicting(identification(x),
                  identification(y))

% definitions of initiate and update:

...

IMPORTING first_refinement
  [Sensor_track, Sensor_track_state,
   System_track, System_track_state,
   correlates, initiate, update]
END second_refinement
```

The second refinement still has parameters for the sensor and system tracks (these are actually object identifiers and are left as implementation detail), for the `Source` of sensor tracks, and for the `Margin` that plays a role in the correlation test. So it is possible to refine this specification even further.

Using the PVS type checker, the second refinement is automatically proved to be type correct, which establishes the correctness of this refinement step. Also note that, since the second refinement is an instance of the first refinement, every theorem proved in the first refinement automatically holds in the second. This holds especially for the verification proof obligations, so no further verification is needed.

5. Conclusion

The general goal of our research is to construct a formal method that supports requirements specification and analysis of industrial-strength systems, with a particular emphasis on command and control systems. Step-wise refinement and tool support are two indispensable means to this end. Using a realistic case study, we have in this paper explored two refinement techniques based on reduction of nondeterminism and instantiation of uninterpreted symbols. Specifications are written in higher-order logic and essentially define a state-transition system that accepts input events and produces output events.

PVS appeared to be well-suited for implementing an environment that directly supports this method. Using the 'importing' mechanism of PVS, the proof obligations to verify the well-definedness of a specification or the correctness of a refinement step can be automatically generated. The interactive proof checker of PVS then assists in constructing the required proofs.

Using uninterpreted symbols, yet incomplete specifications can be verified, or validated by means of putative theorems. The advantage of this technique is that, when later refining these specifications by instantiating some of the uninterpreted symbols with actual definitions, all proven properties of the original specification remain valid for the refined specification.

The designers of PVS at SRI International already advocated the use of PVS during the early phases of system engineering. E.g. [6] describes the use of strong type checking, completeness and consistency checking using powerful decision procedures and model-checking of PVS for requirements engineering. This has been applied in several industrial applications. For instance, [1] describes four case studies in which requirements for new flight software subsystems on NASA's Space Shuttle were analyzed. The size of the analyzed specifications ranges from 20 to 110 pages of informal description. Analysis included reachability analysis using the state exploration tool Murphi and theorem proving with PVS.

Another important issue in the requirements specification and analysis of large-scale systems is the construction of modular specifications. In general it should be possible to decompose a specification into logical components. These do not necessarily coincide with the physical components of the system. The physical decomposition of the system is typically decided in a later design phase. Each logical component should be defined by a number of specification blocks, and it should be possible to verify that the composition of the various components is consistent. In [3] we proposed a method based on temporal logic that supports modular specification and analysis. It remains future research to determine how both methods can be integrated into a unifying framework, capturing modularity and refinement.

References

[1] J. Crow and B. Di Vito. Formalizing space shuttle software requirements: Four case studies. *ACM Transactions on Software Engineering and Methodology*, 7(3):296–332, 1998.

[2] 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.

[3] J.C. van de Pol, J.J.M. Hooman, and E. de Jong. Modular formal specification of data and behaviour. In K. Araki, A. Galloway, and K. Taguchi, editors, *Proceedings of the 1st International Conference on Integrated Formal Methods*, pages 109–128, York, UK, 1999. Springer.

[4] J.C. van de Pol, J.J.M. Hooman, and E. de Jong. Requirements specification and analysis of command and control systems. Technical Report 99-09, Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands, 1999.

[5] M.P.E. Heimdahl and N.G. Leveson. Completeness and consistency in hierarchical state-based requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, 1996.

[6] J. Rushby. Calculating with requirements. In *3rd Symposium on Requirements Engineering*, pages 144–146. IEEE, 1997.

[7] S. Owre, J.M. Rushby, N. Shankar, and F. Von Henke. Formal Verification of Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, 1995.

[8] 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.

[9] 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, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer Verlag, 1996.