Requirements Specification and Analysis of Command and Control Systems

Jaco van de Pol (CWI, Amsterdam)
Jozef Hooman (KUN, Nijmegen)
Edwin de Jong (Hollandse Signaalapparaten BV, Hengelo)

August 25, 1999

Abstract

This report presents a method for formally specifying and analyzing requirements specifications of command and control systems. In this method, a specification consists of a number of specification blocks, each specifying a particular aspect of the system. The main blocks are:

- Enumeration of input and output events with data.
- Description of state variables, constrained by an invariant.
- Mapping of input events to state transitions.
- Mapping of state changes to output events.

Due to the latter mappings, the relation between input and output events is specified indirectly. An input event causes a particular state transition. A state change can be the trigger for a number of output events. The specification blocks are mapped onto a machine model, which is an extension of Mealy Machines. This gives the specification a formal basis.

A formal specification can be rigorously analyzed. We distinguish verification and validation. The verification consists, besides type checking, of proving a number of proof obligations. These obligations address consistency and completeness of the specification. Validation is mainly supported by proving formal challenges of a specification. We suggest induction over the set of reachable states as a powerful proof principle.

The specification language and theorem prover PVS (prototype verification system) is proposed for tool support. It is explained how the specification blocks can be expressed in PVS' language. The specification is type checked by PVS, possibly resulting in a number of conditions to be checked. The specification is automatically mapped onto the machine model. Moreover, the verification proof obligations are automatically generated. The formal challenges can also be expressed in PVS. The type correctness conditions, the proof obligations and the formal challenges, are proved using the PVS theorem prover. PVS supports the proof construction by a number of built-in proof strategies, but in principle, theorem proving is interactive.

We evaluated this method on a realistic system. This is a subsystem of an existing command and control system, viz. track fusion. The various requirements are formalized, and then verified. The specification is built in three stages, where each next stage is a refinement of the previous one. The first refinement has an informal status only, but the second refinement is formal.

Finally, we indicate the directions for future work. The method can be completed by adding modularity mechanisms, and by fine-tuning the relation between input and output events by some protocol description. We expect that this method is fruitful in the design steps, following the requirements specification. To this end, the basic notions of equivalent specifications and refinements have to be studied.

Preface

This report is not only a technical report, presenting a number of scientific and engineering results; in addition it is the final report of an enjoyable two-and-a-half year lasting project at Eindhoven University of Technology. The first author started this research on October 1, 1996, two months before his graduation.

The other authors in a sense started earlier, by getting the research proposal granted. I am indebted to them for this effort. The collaboration was very pleasant, bringing me in touch with the fine fleur of industrially motivated science (Jozef Hooman) and scientifically oriented industry (Edwin de Jong). This continuous cooperation, which was both pleasant and inspiring, had an immense influence on the final contents and form of this report.

I like to acknowledge the support of SENTER (part of the ministry of economical affairs) and Hollandse Signaalapparaten BV, for financing the ORKEST project, of which my position was a part. The other partners were situated at the universities of Amsterdam (UvA) and Groningen (RUG). It was the Eindhoven leg that focused on requirements specifications.

I would like to thank other people as well. First of all Prof. Dieter Hammer, who supervised the Eindhoven part of the project, for his stimulating advice in the project meetings. Further Paul Dechering and Martin Streng from Signaal, who were always interested and willing to share their knowledge. Also the participants in the ORKEST meetings, especially Rix Groenboom (RUG) from Groningen. Finally, Roel Bloo, involved in another research collaboration between Signaal and TUE. I thank him for the lengthy and deep discussions on how systems should be specified and developed, and on any other conceivable practical or theoretical issue.

I hope that the various collaborations will continue in other forms. I intend to continue this research in such a way that it supports industry in the design of correctly functioning systems, and deepens the fundamental understanding of software systems by raising the right questions.

Jaco van de Pol Amsterdam, August 5, 1999.

Contents

Chapter 1

Introduction

The goal of our research is to compose a method for formalizing and analyzing requirements specifications of command and control systems, and to evaluate this method on a realistic system. The method should result in requirements specifications of better quality, that allow to predict the functionality and behaviour of the system under construction. The effect is that potential errors are detected very early in the design process of a system.

In selecting or composing this formal method, the following issues should be taken into account: The method should support the construction of abstract and readable specifications, and it should be scalable to industrially relevant systems in the intended application domain. Finally, it should support verification of desired properties, like consistency and completeness.

1.1 Command and control systems

The general task of a command and control system is to support a team of operators in monitoring and controlling the environment in order to accomplish a mission. Commonly, these systems support tasks like navigation, observation, communication, defense, and training. Similar applications include traffic management systems, air traffic control systems and process control systems.

Command and control 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 the 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 analyzing corrective actions. These actions are scheduled for execution, by assigning a time-frame and sufficient resources, and eventually executed via the actuators.

Command and control systems are typically large and complex, whereas the standards on correctness, reliability and availability are high. Hence it is a difficult and error-prone task to build such systems. It is important to be able to manage the time and costs needed to develop a particular system. Too often fatal errors are detected on testing a system that has been built already. In that case parts of the development must be reiterated, which results in additional and usually unpredictable costs. Of course, the damage of errors detected after delivery is even more disastrous, encompassing severe economical as well as social aspects.

1.2 Requirements engineering and formal methods

We refer to [SS97] for a general treatise on requirements engineering. The requirements specification is the starting point of the development process. To have a solid basis for system development it is important to have an unambiguous and truthful description of the requirements on such systems. From the previous section, it appears that it is preferable to detect errors at an early stage of the development process, viz. in the requirements specification phase. Errors in the specification propagate to all later phases in the development, until they are detected. In fact, it is well known from the literature, that errors that are made early and detected late, are relatively expensive to repair. Therefore, the "quality" of the requirements specification is an important issue.

This "quality" has many aspects. Surely, the specification should be the true expression of the requirements of the users. Various validation techniques have been developed to assess that the specification is according to this intention, like inspection, simulation and rapid prototyping. However, because there is no authoritative document against which the specification can be checked, validation, either by users or by domain experts, is always partial.

Another way to assess the "quality" of the specification, is to find intrinsic properties that a good specification should have. Some desirable criteria that a requirements specification has to meet are: unambiguity, consistency, completeness and readability. It is also desirable that a specification is abstract, which means that it is not biased to any particular solution, but focuses on the problem statement only. Potential errors in the specification can be detected by analyzing whether the desirable criteria are met.

A rigorous analysis is only possible, if the specification is precise and unambiguous. Because informal specifications, which are written in natural language and illustrated by diagrams, tend to be ambiguous and imprecise, we think that *formal* specification techniques help to detect errors early. It is quite commonly agreed nowadays that formal methods can be helpful in the earliest phase of system development. Using a formal specification language ensures that the specification is unambiguous. A formal semantics provides for a mathematical model of the specified system, which can be analyzed with mathematical rigour. Thus properties of the specification itself, like consistency, can be verified formally. Finally, powerful tools, like theorem provers, are available to make this analysis tractable.

1.3 Outline of this report

Chapter 2 introduces and explains our specification method. A tool-independent explanation is given in 2.1–2.3. Section 2.4 is devoted to tool support for the method. We propose to use the theorem prover PVS [ORSH95, SSJ+96] (Prototype Verification System) in order to support the method. The explanation in this chapter is deliberately kept global.

For a detailed explanation, we refer to the case study in Chapter 3. This chapter follows the method, serving both as an illustration and as an assessment of the proposed method. Details of PVS that are required to understand the specification, are explained on the fly. Sections 3.2–3.4 contain the full specification; section 3.5 is devoted to its analysis.

Chapter 4 contains two refinements of the specification given in Chapter 3. The purpose is (1) to illustrate some alternative formalizations of certain concepts; and (2) to show how a specification can be refined. Both refinements consist of a specification and its analysis.

We evaluate our method and enumerate some possibilities for future work in Chapter 5. Finally, the appendices give full specifications and proofs, templates to use the method for another application, and an overview of related literature.

Chapter 2

Specification and Analysis Method

First, the underlying machine model is explained (section 2.1). From this a concrete specification format is derived (section 2.2). It is also explained how a specification can be analyzed (section 2.3). In section 2.4 we explain how PVS can be used for tool support.

2.1 Machine model

The interface between the system and its environment, is modeled by *events*. One may distinguish input events (where the environment has the initiative) from output events (for which the system is responsible). Typical input events are sensor measurement reports and operator commands. Typical output events are display commands and control signals to actuators. The specification has to define the relationship between input and output events.

For most applications, a direct specification of this relationship is not feasible. For this reason the *state* is introduced as an auxiliary means to specify the relationship between input and output events. On the basis of the input events, the system obtains information on the current state of affairs in the environment. This information is represented by the state of the system. The state is used to decide which output events occur.

We take the following modeling assumption: Input events are atomic, so they occur instantaneously, one at a time. An input event causes a state transition, which also takes place instantaneously. A state transition may trigger (cause) output events. Similar assumptions are made in Mealy's finite state machines, introduced in [Mea55]. As a starting point, we introduce these Mealy machines, and then we present our machine model as a modification. Recall (cf. [HU80, p. 43]) that a Mealy FSM is a six-tuple $(I, O, S, \delta, \lambda, s_0)$, where

- I is a finite set of input symbols;
- O is a finite set of output symbols;
- S is a finite set of states;
- $\delta : S \times I \to S$ is a state transition function;
- $\lambda : S \times I \to O$ is an output function;
- $s_0 \in S$ is the initial state.

The machine starts in s_0. If during execution, the machine is in some state $s \in S$ and it gets input symbol $i \in I$, then the next state will be $\delta(s, i)$. During the transition, the machine emits output symbol $\lambda(s, i)$. Note that the machine is always ready to accept any input, is deterministic, and produces one output at a time.

Such machines stem from language theory, and it appears that all kinds of modifications yield equivalent formalisms, such as associating output events with states rather than transitions, allowing non-determinism, empty steps etc. However, although all these variations are immaterial from a theoretical point of view, in practice these modifications matter. We now modify the machine model above, in order to make it more useful as a model for software specifications. Four modifications are needed, which are motivated below:

- The sets I, O and S may be *infinite*;
- The transition function δ may be defined *partially* and/or *non-deterministically*.
- On each transition a *set* of output symbols is emitted, and the occurrence of these output symbols depends on state *transitions*.
- A *set* of possible initial states is allowed.

These modifications are introduced for pragmatic reasons. Allowing an infinite set of states increases the expressive power of the model considerably. The state can be viewed as an assignment of values to state variables; this corresponds to what is known as *extended* finite state machines. Using infinite sets of events allows events to carry data parameters.

A certain degree of implementation freedom should be allowed in requirements specifications. This avoids the specification of containing irrelevant requirements and details. Therefore, the state transition function may be non-deterministic. Design decisions can be taken in the implementation, which decrease the level of non-determinism. For similar reasons, it can be unnatural to require a unique initial state. It is also allowed that the system is not ready for some input event at certain moments. We will come back to this later.

In command and control applications, output events are typically triggered by state *transitions*, and not directly coupled to input events. As an example, consider the receipt of some sensor measurement (input event), which leads to updating the kinematics of the corresponding system track (state transition). It may turn out that by this change, the track is now leaving the air lane which it was supposed to follow. Leaving the airlane (a state change) should lead to an IFF (identification friend or foe) interrogation (an output event). Clearly, it is much more natural to couple the IFF interrogation to "leaving the airlane" than to the "receipt of a sensor measurement".

To formalize these ideas, we could have $\delta : S \times I \to \wp(S)$ and $\lambda : S \times S \to \wp(O)$. However, we choose the following equivalent formulation, which is symmetric and turns out to be more handy in actual specifications. The proposed machine model is a six tuple: $(I, O, S, \Delta, \Lambda, S_0)$, where

- I is a (possibly infinite) set of input events;
- O is a (possibly infinite) set of output events;
- S is a (possibly infinite) set of states;
- $\Delta : I \to \wp(S \times S)$ is the state transition function;
- $\Lambda : O \to \wp(S \times S)$ is the output trigger function;
- $S_0 \subseteq S$ is a nonempty set of possible initial states.

This is to be interpreted as follows. Initially, the system is in some state $s \in S_0$. Assume that during execution, the system is in some state $s \in S$, and that input event $i \in I$ occurs. The system then moves (non-deterministically) to some state $\mathtt{t}$, such that $(s, t) \in \Delta(i)$ (if it exists). During this transition, all output events o, for which $(s, t) \in \Lambda(o)$ are triggered. If such a t cannot be found, the system is blocked (deadlock) which is an undesirable situation. This gives an implicit precondition on the occurrence of input events.

Remark. This gives a precise mathematical model. The physical interpretation is not completely clear yet. One extreme interpretation could be that all triggered output events actually occur *before* the next input event occurs. This is quite unnatural, and even impossible (or unnecessarily hard) to implement in distributed systems. The other extreme interpretation is that output events are merely *caused* by a state transition, and should happen *eventually*. The latter interpretation is preferable, although it would probably be useful to add time bounds to such a specification.

This issue has a major impact on the notions "equivalence of machines", and "simulations between machines". However, we feel that the outcome of current research on the design trajectory is needed before these issues can be resolved. See also the discussion on future work in Chapter 5.

2.2 Concrete Specifications

2.2.1 Extensions to the machine model

For large-scale applications such as command and control systems, it is not feasible to specify a system in terms of the above machine model directly. To this end, we propose a more concrete specification format. We first add some parts to the machine model, which are superfluous from a theoretical point of view, but which greatly improve the conciseness, abstractness and analyzability of a specification. These ingredients are:

- $Inv \in \wp(S)$, the invariant, containing integrity constraints on the state.
- $Pre \in I \to \wp(S)$, the precondition of the input events, representing the assumptions on the environment.

The advantage of having invariants is, that we need not specify for each state transition how a constraint is maintained. Instead, we can directly specify that a constraint should always hold. In this way, the specification tends to become less operational and more declarative. Another advantage is that we cannot forget such constraints in one of the state transitions, which can easily happen if we have to repeat it.

We mentioned earlier that Δ imposes an implicit precondition on the input events. These preconditions can be seen as requirements that the environment of the system should meet (as long as the environment meets these conditions, the system won't block). As we generally cannot construct the environment, it is important to have an explicit statement of the assumptions on the environment. In this way, the integration of a system in a certain environment can be verified to work.

Formally, the invariant and the precondition can be encoded in Δ. A transition from state s to t by input event i can happen if and only if $s \in Pre(i)$ and $(s, t) \in \Delta(i)$ and $t \in Inv$ hold. Alternatively, the invariant could be seen as a restriction on S rather than on Δ.

2.2.2 Building blocks of a specification

In the concrete specification, eleven building blocks can be distinguished. Together, these building blocks define the ingredients introduced above, and can be mapped onto the abstract machine model. In the following, we divide these blocks into sections, and give a natural order to pass through the specification process. Of course it is highly unlikely that each step is always completed before proceeding with the next step. This ordering should be seen as a rule of thumb, not as an obligation.

A. Static interface. We want to start with the interface, i.e. the input and output events, because this defines the border of the system, which is one of the major tasks of a requirements specification. Recall that events may carry data. The type of these data have to be defined too, in the form of a number of (externally visible) basic entities. So the static interface is defined by the following three blocks:

1. Definition of basic entities
2. List of input events with data parameters (I)
3. List of output events with data parameters (O)

B. Information model. Next, we turn to the internal aspects of the system, starting with the information that is maintained in the internal state. This state reflects the system's knowledge on its environment. It is advisable to model this information in an implementation independent way. The ideal situation would be that the information model could just be derived from the application domain. The description of the internal state consists of:

4. List of state variables with types (S)
5. Specification of integrity constraints (Inv)
6. Definition of allowed initial states (S_0)

C. Mapping input to output. Then, we couple input to output events, via the state. For each input event, we have to specify the state transition caused by it. Similarly, for each output event we specify by which state change it is triggered. The mappings from input events to state transitions, and from output events to triggers then consist of a simple enumeration.

7. Specification of state transitions (Δ_i)
8. Specification of triggers (Λ_o)
9. Mapping of input events to state transitions (Δ)
10. Mapping of output events to triggers (Λ)

D. Environment. Finally, the preconditions can be added. Although the precondition in fact belongs to the system interface, we put it here because it can be derived from Δ and Inv.

11. Definition of assumptions on the environment (Pre)

2.3 Analysis

For the analysis, we distinguish 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 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. In fact, *type correctness* is a part of the verification. However, because of its importance we deal with it separately. So the analysis process consists of the following activities:

- Parsing and type checking
- Verification
  - Existence of an initial state satisfying the invariant
  - Totality of state transitions w.r.t. the preconditions
- Validation
  - Inspection
  - Formal challenges

We next discuss each of these activities in more detail.

2.3.1 Parsing and type checking

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 logical language, type checking can also guarantee semantical completeness, for instance totality of functions, exhaustiveness of case distinctions and tables etc.

Also, in most formal languages that are used in theorem provers, the addition of well typed definitions is guaranteed to be a conservative extension, so if we refrain from using axioms, type checking guarantees logical consistency.

2.3.2 Verification

It is important to check 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 combination of all integrity constraints is consistent, since we found a model.

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

Next, recall that on occurrence of input event i in state s, the state changes to some t, such that $(s,t) \in \Delta(i)$ and $t \in Inv$. The system is blocked if such a t doesn't exist. So there is an implicit precondition on the occurrence of i in state s, viz. $\exists t.\, t \in Inv \wedge (s,t) \in \Delta(i)$.

This formula is quite complicated in practice, and it is also hard to grasp its meaning, due to the existential quantifier. Therefore, we have required the specification of an explicit precondition, $Pre(i)$. This precondition needs only be an approximation of the implicit precondition, so it can be much simpler. In order to verify that the system specification is

on the safe side, we have to prove that $Pre(i)$ implies the existence of t above, so it is stronger than the implicit precondition. Note that we can safely assume that s satisfies the invariant. By proving this, we establish deadlock freedom in any environment that guarantees the preconditions. Moreover, it shows that the state transitions and the invariants are consistent (under assumption of the precondition), because the existence of transitions that satisfy both is proved. So we have:

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 don't yet have an obligation to check that the specification of the output events is reasonable. As in [HL96], it could be required that the state transitions are deterministic. We think, however, that this requirement is too strict in the setting of distributed systems.

2.3.3 Validation

One form of validation is inspection by domain experts. This requires that the specification is well documented. Another form that is only possible for formal specifications, is to *challenge* the specification by proving that it guarantees a number of expected properties. This really is a form of *testing* the specification.

Inspection

In order to enable inspection, the specification should be accessible. This is achieved by the building blocks, giving the specifications a fairly standard layout. But of course, formal languages tend to be less readable than English prose, although ...

Another way to support inspection is to use a *literate specification* paradigm. Following the literate programming tradition, this is a form in which formal parts and informal explanation are intertwined. Special tools can extract the formal part, in order to formally analyze it, or typeset the whole document, for inspection. In a previous report, we have used `noweb` for this purpose [Ram94]. Another advantage of such a tool is that the program text in the source files and in the documentation cannot diverge. Of course, the informal and formal parts in the text should be kept consistent by hand.

The literate specification approach supports validation by inspection in two ways:

- A domain expert can validate the informal parts, and glance at the formal part whenever ambiguities arise.
- A software expert can validate the formal parts, by comparing them with the informal parts.

Putative theorems

A way of validation which is only possible for formal specifications is to challenge it by putative theorems. These should be proved to hold for/follow from the specification. The idea is to check whether certain properties that intuitively are expected to follow are really implied by the specification. This can reveal numerous errors. The exact form of these theorems is dependent of the system under specification, so this is an ad hoc method.

In order to prove the putative theorems, we need proof principles. One of the most powerful principles is induction over the reachable states of the system. A state is reachable, if the system can reach it from the initial state by inputs satisfying the preconditions. Formally, it is the smallest set $\mathcal{R}$, satisfying:

- $S_0 \cap Inv \subseteq \mathcal{R}$.
- If $s \in \mathcal{R}$, $s \in Pre(i)$, $t \in Inv$ and $(s,t) \in \Delta(i)$, then $t \in \mathcal{R}$.

The induction principle connected with this inductive definition allows us to prove that a certain property P holds for all reachable states. Basically, it reduces the task of proving P for all reachable states, to the more feasible tasks of proving P for the initial states, and proving that P is maintained by all possible transitions. Formally:
From

- $\forall s \in S_0.\, P(s)$; and
- $\forall s,t \in S, i \in I.\, \Big(s \in \mathcal{R} \wedge s \in Pre(i) \wedge t \in Inv \wedge (s,t) \in \Delta(i) \wedge P(s)\Big) \Rightarrow P(t)$,

we may derive $\forall r \in \mathcal{R}.\, P(r)$.

2.4 Tool support

We see tool support as an indispensable means to achieve practical applicability and scalability of our requirements specification approach. Theorem provers can be effectively used in the analysis of a specification. Numerous errors can be found by parsing and type checking the specification, and properties of the specification can be checked by computer-aided verification of certain theorems.

2.4.1 PVS

In order to have tool support at this experimental stage of research, we use an existing general-purpose theorem prover. We have implemented our ideas in PVS (Prototype Verification System) [ORSH95, SSJ+96]. PVS is a specification language integrated with support tools and a theorem prover, developed at SRI, Stanford.

The specification language is 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, axioms, and theorems. These can be made available in other theories, by the importing-mechanism. A theory can also make assumptions on its parameters.

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.

The TCCs, together with the 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 provides commands to apply the proof rules, via an EMACS-interface. 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.

Finally, PVS provides additional tools, for instance to generate status reports of the theorems that still have to be proved, to edit and rerun proofs after small modifications of the specification, to pretty print theories and proofs and to visualize proofs and theory trees.

In principle there is no fundamental reason why PVS should be chosen. We have some experience with Coq and Isabelle, and these tools can surely be used for the same purpose. A number of advantages for PVS could be:

- It has been used for other industrial case studies (in fact it is developed in close collaboration with NASA)
- It is relatively easy to learn, certainly in comparison with the other mentioned provers (the main problem seems to be to learn EMACS, the editor used for user interaction).
- The specifications are quite readable, due to common notation from functional programming, tabular notation and the possibility to use LaTeX conversions.
- The language is expressive, due to higher-order logic, record types, subtypes, dependent types and abstract datatypes.
- It has built-in natural and real numbers, and it is equipped with a library, containing among others lists, sets, orderings, ordinals, infinite sequences etc. with many useful theorems.
- It has quite powerful automatic proof search facilities.

We think that the use of more automated theorem provers (such as resolution provers, like OTTER) will not be successful, because automation is achieved at the cost of simplifying the language (for instance to first-order, or even propositional logic). Such languages are not expressive enough for requirements specifications.

An advantage of Coq and Isabelle could be that they have true polymorphism, which can be helpful in the meta theory of specifications. Moreover, if it comes to writing advanced proof tactics, Isabelle might be preferable above PVS.

2.4.2 What is supported

PVS supports the proposed specification method in the following ways:

- It provides the language for the specification;
- It is used for parsing and type checking the specification, automatically generating TCCs where needed;
- It is used to automatically generate the verification proof obligations of section 2.3.2;
- It supports the interactive proof of the TCCs, the verification proof obligations, and the user provided formal challenges.

In our approach, a specification consists of a number of parts, corresponding to the blocks of section 2.2. These parts define the components $(I, O, S, \Delta, \Lambda, init, Inv, Pre)$ for the machine model. We then import a generic theory, which is fixed for all specifications (see appendix A.1 and section 3.5.1). This theory has the above mentioned parameters, and it assumes that the verification proof obligations hold. On importing this theory, we have to prove that these assumptions hold for the actual parameters, viz. the $I, O, \ldots$ of the specification. This general theory also defines the reachability predicate. Meta theory on specifications could be developed in this theory.

2.4.3 Requirements in PVS

Appendix A.2 contains an extensive template, which can be used as a basis for writing concrete specifications in PVS. It contains the various specification blocks, and makes a number of basic definitions. However, because the reader might not be familiar with PVS, we don't explain this template here. Instead, we provide a gentle introduction to PVS by example. The next section contains a complete specification in PVS. The specification language is explained on the fly.

Chapter 3

Case study

To illustrate and evaluate our approach, we formally specify and analyze the requirements for a subsystem of a realistic command and control system. The specifications in this report are based on [PHJ98], which was in turn derived from the informal specification of an existing command and control system. We now adapt the requirements from that paper to track fusion, instead of track joining. Another modification is that we now add output events.

We first informally describe track fusion. Then we go into the specification. Sections 3.2–3.4 correspond to the blocks of the specification method. Section 3.5 describes the analysis of this specification. Each concept is first informally introduced, then the formalization in PVS follows. Where needed, some remarks on the PVS language follow in italic font.

The specification in this section is quite global. A number of requirements is missing, and a number of details is missing. In section 4 we introduce two refinements of the specification, the first (4.1) introduces more requirements on the operational behaviour of the system, and the second refinement (4.3) adds detail.

Track fusion

A track is a description of a real-world object, reporting on e.g. measured position, velocity, identification etc. Tracks occur on (at least) two levels:

- Sensor tracks, as reported by a sensor.
- System tracks, as generated by track fusion.

An object in the real world may be detected by various sensors. Since sensors are not perfect, this results in slightly different *sensor* tracks. In order to present a global and coherent picture, sensor tracks should be *fused* into a single *system* track, if they represent the same real-world object. One of the tasks of track management is to derive and maintain the set of system tracks. Certain correlation criteria define whether tracks are considered to represent the same object, or not.

The various sensors can initiate new sensor tracks, and update or wipe existing sensor tracks; this is the input to the system. An abstract picture is compiled, containing the current sets of sensor- and system tracks, their relationship, and the derived kinematic information. The system has to report the derived system tracks to the operator. We distinguish between the initiation, an update, or the deletion of a system track. The system also has to generate a warning to the operator whenever a sensor track decorrelates from a system track.

In the sequel, we formalize the relationship between sensor- and system tracks, and we introduce the manipulations on them in a rather global way. The exact kinematic calculations will not be specified. The formalization is along the lines of the method described in section 2.2, but the emphasis of the first specification lies on the information model.

3.1 Static interface

3.1.1 Types of basic entities

We start with declaring some basic types, of which it is decided that they need no further formalization. Types `Sensor_track` and `System_track` can be seen as identifiers for sensor and system tracks. Their actual values is an implementation issue. The types that define the actual sensor and system track states are important for the specification, but since we don't yet specify kinematic computations, these details are deferred to later refinements.

```
Sensor_track,System_track: TYPE
Sensor_track_state, System_track_state: TYPE+
```

We use PVS uninterpreted type declarations, using the keyword `TYPE` *or* `TYPE+`*. The latter indicates that the type is non-empty. Uninterpreted types are guaranteed to be disjoint with each other and with any other type, i.e. they don't overlap.*

3.1.2 Input events

We have three different input events: `new`, `update` and `wipe`, with the purpose of initiating, updating and deleting sensor tracks. To each event, certain data parameters are attached. For instance, for wiping a sensor track we need to know which track is wiped. For the other events we additionally need the current state of that track. The set of input events can be defined as an abstract datatype. In this way we can have finitely many event names, and each event name is coupled to a fixed number of parameters, with a possibly infinite range.

```
IEvents: DATATYPE
BEGIN
  new(sn:Sensor_track,s:Sensor_track_state): new_sens?
  update(sn:Sensor_track,s:Sensor_track_state): update_sens?
  wipe(sn:Sensor_track): wipe_sens?
END IEvents
```

We used a powerful PVS construct: abstract data types, distinguished by the keyword `DATATYPE`*. An abstract data type definition defines a new type (here* `IEvents`*), disjoint with all other types, together with constructors, destructors and recognizers. The* constructors *correspond with the event names, here* `new`*,* `update` *and* `wipe`*. They exhaustively enumerate the elements of the data type. Constructors can take arguments corresponding to the data parameters, whose types are listed in an argument list. The names of the destructors are declared in this argument list too. The* destructors *correspond to the argument names. They serve as accessors to the arguments of the constructors. We typically have:* `sn(new(x,y))=x` *and* `s(new(x,y))=y`*. Finally, we get* recognizers*, which are predicates to recognize the top symbol of a term of the data type. These can be useful in tests, or in subtypes. Typically,* `wipe_sens?(wipe(sn))` *equals* `TRUE`*.*

In PVS, `[s->t]` *denotes the type of functions with domain* `s` *and range* `t`. `bool` *is the built-in type of boolean values.* `(new_sens?)` *denotes the subtype corresponding to the predicate* `new_sens?`. *Subtypes are explained in more detail in section 4.1.2.*

Armed with these types, we can indicate the types of the recognizers, constructors and destructors:
`new_sens?:[IEvents->bool]`; `new:[Sensor_track,Sensor_track_state->(new_sens?)]`; `sn:[(wipe_sens?)->Sensor_track]`. In fact we use overloading, because we introduce three different functions `sn`.

3.1.3 Output events

The definition of the output events is similar. We introduce three different output events: `new`, `update`, and `wipe`, which correspond to the creation, update or deletion of system tracks. The operator should be warned whenever a pair of sensor track and system track decorrelates, which we model by the output event `warn`.

```
OEvents: DATATYPE
BEGIN
  new(tn:System_track,t:System_track_state): new_sys?
  update(tn:System_track,t:System_track_state): update_sys?
  wipe(tn:System_track): wipe_sys?
  warn(sn:Sensor_track): warn_sys?
END OEvents
```

3.2 Information model

3.2.1 State variables

We model the global state as a record with the following components: The set of sensor tracks received until now; the set of system tracks derived from these; a relation indicating which sensor tracks are currently joined to which system tracks; and a function which returns for each system track its state. Note that the state vectors of the sensor tracks are not kept in the global state.

```
State: TYPE =
  [# sensor_ids: setof[Sensor_track],
     system_ids: setof[System_track],
     system_info: [System_track->System_track_state],
     joined: pred[[Sensor_track,System_track]]
  #]
```

Here we have used the record construction (`[# ...#]`*), which denotes a tuple (or Cartesian product) with named fields. Each field has its own type; the order of the fields is immaterial. The field names can be used as accessors to the components of the records: Given* `X:State`, `joined(X)` *for instance denotes the fourth component. We now explain the types of the components.*

The type `[T->S]` *denotes the type of total functions from* `T` *to* `S`. *Product types (pairs) are denoted by* `[T,S]`. *We have used the type constructors* `setof[T]`, *denoting the type of sets over* `T` *(powerset), and* `pred[T]`, *denoting the type of predicates over* `T`. *In fact, both are just abbreviations of* `[T->bool]`, *so sets and predicates are represented by their characteristic*

functions. By definition, `x` *is a member of* `S` *if and only if* `S(x)` *holds. Note that* `pred[[T,S]]` *denotes predicates over pairs, which can be identified with binary relations. At this point we profit from higher-order logic, where functions, and hence sets and predicates, are first class citizens, so they can appear as components in record types.*

Alternative

It would also be possible to see the sensor track identifiers as a part of the sensor track state, and similarly for system tracks. The joined relation could then be defined as a relation between track states, and the function `system_info` would be superfluous. Although this simplifies the structure of the global state, a complication would arise later, because after a change in the kinematics of a sensor track state, the corresponding pair in the joined relation has to be updated too. In the current definition the pairs in the joined relation are stable under kinematic changes.

3.2.2 Invariants

By formulating a number of invariants, or integrity constraints, we can fix a number of global properties of the system, without indicating how they should be maintained. In this section, the relation between sensor and system tracks is established via constraints. In particular, we require that the joined relation is a surjective, total function from the current set of sensor tracks to the current set of system tracks. In this way, a system track can be perceived as a group of one or more sensor tracks.

Variable declarations

In the sequel a number of variables is used. In order to avoid that we have to indicate the type of these variables again and again, we can declare their types once and for all:

```
s,s1: VAR Sensor_track_state
sn,sn1: VAR Sensor_track
t: VAR System_track_state
tn,tn1,tn2: VAR System_track
ie: VAR IEvents
oe: VAR OEvents
X,Y: VAR State
```

The keyword `VAR` *marks these declarations as variable declarations, which should be distinguished from constant declarations. Variables have no global value, they only appear in function definitions (as in* `f(x,y) = ...`*) or as bound variables (as in* `FORALL x: ...`*). The type of* `x` *and* `y` *in these examples are derived from the variable declarations. A frequently occurring error in PVS texts is to drop the keyword* `VAR`*. This would result in a constant declaration.*

Constraints

We now come to the definition of the invariant properties. The invariant is represented as the conjunction of a number of constraints. A constraint is a property of the state, so it can be defined as a function `[State->bool]`. Here we define the following constraints:

The joined-relation only contains tracks that actually are present in the current state.

```
constraint1(X):bool = FORALL sn,tn :
   joined(X)(sn,tn) => sensor_ids(X)(sn) & system_ids(X)(tn)
```

This is a constant declaration, defining the constant (or function) `constraint1` *of type* `[State->bool]`*. In higher-order logic, formulae are just data terms of type* `bool`*. Note that* `joined(X)` *is a predicate, which can be applied to a pair* `(sn,tn)`*, yielding a boolean.*

We can use the well-known connectives and quantifiers of predicate logic, using the following symbols:

Symbol	*Meaning*
`TRUE`	*truth*
`FALSE`	*falsehood*
`NOT`	*negation*
`&` *or* `AND`	*conjunction*
`OR`	*disjunction*
`=>` *or* `IMPLIES`	*implication*
`FORALL`	*universal quantification*
`EXISTS`	*existential quantification*
`exists1!`	*unique existence*

We continue with the constraints: Each sensor track in the state is joined to precisely one system track:

```
constraint2(X):bool = FORALL sn:
  sensor_ids(X)(sn) => exists1! tn: joined(X)(sn,tn)
```

To each system track in the state at least one sensor track is joined:

```
constraint3(X):bool = FORALL tn:
  system_ids(X)(tn) => EXISTS sn: joined(X)(sn,tn)
```

As mentioned before, the invariant is just the conjunction of all constraints:

```
Invariant(X):bool =
  constraint1(X) & constraint2(X) & constraint3(X)
```

3.2.3 Initial states

We can now define the set of allowed initial states of the system. In an initial state, the sets of sensor tracks is empty. Note that the invariant then restricts the set of system tracks and the join-relation (cf. exercise 1 on page 27).

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

`empty?` *is defined in the prelude as a predicate on sets.*

The "result" of the information model is a type `State`, with a predicate `Invariant` on it and an element `initial:State`. In the analysis (section 3.5) we show that there exist at least one initial state satisfying the invariant.

Alternative

Instead of having separate invariants, we could incorporate the invariant properties directly into the state from Section 3.2.1. Using subtypes, we can view the result as the type {`X:State | Invariant(X)`}, denoting the type of those states that satisfy the invariants. This type can be abbreviated by `(Invariant)`. So the initial state could then be defined as `initial : (Invariant) = ...`. This would of course generate a proof obligation. See section 4.1.2 for more on subtypes.

3.3 Mapping input to output

Recall that both state transitions and triggers, are represented as binary relations $R(s,t)$, where s denotes the state before the change, and t the state after.

3.3.1 State transitions

For each type of input event, we next define the corresponding state transition. When a new sensor track is reported, it might be joined to an existing system track, provided certain correlation criteria are met. Conversely, it might be that, after receiving a sensor track update, the sensor track and the system track to which it was joined decorrelate. This is decided by a set of decorrelation criteria. In order to obtain a stable system, the decorrelation criteria are usually not exactly the negation of the correlation criteria.

In order to avoid detailed calculations at this level of the specification, we introduce the correlation and decorrelation criteria as uninterpreted relations between a sensor and a system track state. However, to avoid arbitrary interpretations of these relations, we add some restrictions on them: It should be possible to initiate some system track on a sensor track, and the correlation and decorrelation criteria should be exclusive.

```
correlates(s,t):bool
decorrelates(s,t):bool

corex   : AXIOM  EXISTS t : correlates(s,t)
cordecor: AXIOM  NOT (correlates(s,t) & decorrelates(s,t))
```

We used `AXIOM`*s, in order to formalize the assumptions on the correlation and decorrelation criteria. An axiom is regarded true by PVS without any proof obligation. The axioms above contain free variables. PVS takes the universal closure of the axioms, so the first axiom really means* `FORALL s : ...` *and the second axiom means* `FORALL s,t : ...`.

There is an important distinction between axiom declarations, and constant declarations of type bool. The latter merely define certain formulae, while axiom declarations additionally assert that these formulae should be considered true.

It is quite dangerous to add axioms to a specification, because they may introduce logical inconsistencies into the specification. Restricting to definitions only, guarantees logical consistency of a specification (provided PVS is consistent). If the added axioms are inconsistent any proofs that are carried out are worthless. In section 4.1 we will explain how axioms can be avoided altogether.

We are now in a position to define the state transitions for the input events. When a new sensor track is initiated, we require that the track is added to the current set of sensor tracks, and that it correlates to the system track to which it is joined.

```
new_sensor_track(sn,s)(X,Y):bool =
  sensor_ids(Y) = add(sn,sensor_ids(X))
& FORALL tn: joined(Y)(sn,tn)
             => correlates(s,system_info(Y)(tn))
```

Note that exactly one system track is joined to `sn` *by the invariant property of Section 3.2.2. By using* `FORALL` *we need not know which system track this is. We use the function* `add(x,S)` *from the prelude, which adds an element* `x` *to a set* `S`*.*

When a sensor track is wiped, we require that it is removed from the current set of sensor tracks. Furthermore, we require that the set of system tracks and the joined relation can only shrink.

```
wipe_sensor_track(sn)(X,Y):bool =
  sensor_ids(Y) = remove(sn,sensor_ids(X))
& subset?(system_ids(Y),system_ids(X))
& subset?(joined(Y),joined(X))
```

We use `subset?(S,T)` *from the prelude, which denotes that* `S` *is a subset of* `T`*. The function* `remove(x,S)` *denotes the set* $S \setminus \{x\}$*.*

Finally, when a sensor track is updated, the set of sensor tracks remains unchanged. We furthermore require that in the new state, the sensor track and the system track to which it is joined don't decorrelate.

```
update_sensor_track(sn,s)(X,Y):bool =
  sensor_ids(Y) = sensor_ids(X)
& FORALL tn: decorrelates(s,system_info(Y)(tn))
             => NOT joined(Y)(sn,tn)
```

3.3.2 Triggers for output events

We next define the state changes that triggers the output events. Given the states `(X,Y)` before and after a state transition, a system track is

- apparently new if it occurs in `Y` but not in `X`.
- updated if the corresponding track state in `X` and `Y` are different.
- wiped if it occurs in `X` but not in `Y`.

Finally, we detect a decorrelation of `sn` if it is joined to different system tracks in `X` and `Y`. We define four triggers accordingly:

```
new_system_trigger(tn,t)(X,Y):bool
=   system_ids(Y)(tn)
  & NOT system_ids(X)(tn)
  & t=system_info(Y)(tn)

update_system_trigger(tn,t)(X,Y):bool
=   system_ids(X)(tn)
  & system_ids(Y)(tn)
  & system_info(X)(tn) /= system_info(Y)(tn)
  & t = system_info(Y)(tn)

wipe_system_trigger(tn)(X,Y):bool
=   system_ids(X)(tn)
  & NOT system_ids(Y)(tn)

detect_decorrelation(sn)(X,Y):bool =
 EXISTS tn1,tn2 :
    joined(X)(sn,tn1)
  & joined(Y)(sn,tn2)
  & tn1 /= tn2
```

The infix symbol `/=` *denotes inequality.*

3.3.3 Mapping of input events to state transitions

The effect of an input event is given by the following table, translating each event to a (non-deterministic) state transition, defined before.

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

We use the `CASES` *construction, which denotes pattern matching on a term of an abstract data type (here* `ie`*). The cases should be exhaustive, or a final* `ELSE` *clause is obliged. A pattern may contain variables, which are local to a line. This construction is very convenient to map events with data to parameterized state transitions.*

3.3.4 Mapping of state transitions to output events

For each output event, we introduce a corresponding state change that triggers that event, by the following mapping:

```
Output_table(oe):[State,State->bool] =
  CASES oe OF
    new(tn,t)   : new_system_trigger(tn,t),
    update(tn,t): update_system_trigger(tn,t),
    wipe(tn)    : wipe_system_trigger(tn),
    warn(sn)    : detect_decorrelation(sn)
  ENDCASES
```

3.4 Assumptions on the environment

The specification is completed by making the assumptions on the system interface explicit. In our example, this is quite easy. It amounts to the following conditions:

- Only existing sensor tracks can be updated and wiped;
- Only fresh sensor tracks can be initiated; and
- There always exist fresh system track identifiers (this is only needed for `new` and `update` events).

It is clear that these are reasonable assumptions. The first and second assumptions are satisfied by any sensible sensor. The third condition requires that the set of identifiers used in the implementation is large enough for the number of tracks the system should be able to handle (actually, this isn't a condition on the environment, of course). The preconditions are defined as a mapping on input events:

```
Precondition(ie)(X):bool =
  CASES ie OF
    new(sn,s)    :   (NOT sensor_ids(X)(sn))
                   & (EXISTS tn: NOT system_ids(X)(tn)),
    update(sn,s) :   sensor_ids(X)(sn)
                   & (EXISTS tn: NOT system_ids(X)(tn)),
    wipe(sn)     : sensor_ids(X)(sn)
  ENDCASES
```

3.5 Analysis

We now proceed with the verification of the proof obligations. In order to proceed with the verification, we have to import the generic PVS theory, which represents the machine model, and generates the proof obligations. In the next section, we describe this generic theory, and in section 3.5.2 we report on the verification of the automatically generated proof obligations.

3.5.1 Importing the machine model

In our approach, the abstract machine model (section 2.1) consists of a number of components, viz. $(I, O, S, \Delta, \Lambda, init, Inv, Pre)$. The concrete specification consists of a number of blocks, defining such components. In the previous sections, we have defined:

- `IEvents`

- OEvents
- State
- Input_table
- Output_table
- initial
- Invariant
- Precondition

We now import a generic theory (machine), which is fixed for all specifications. This theory has the following parameters: I, O, S, Imap, Omap, init, Inv and Pre, corresponding to the components mentioned above. As assumptions, it contains the verification proof obligations of section 2.3.2. On importing this theory, we have to prove that these assumptions hold for the actual parameters, viz. the IEvents, OEvents,... of the concrete specification. The general theory also defines the reachability predicate. Finally, it proves that the invariant holds for all reachable states. The general theory reads as follows:

```
machine[  I,O,S: TYPE,
          Imap: [I -> pred[[S,S]]],
          Omap: [O -> pred[[S,S]]],
          init: pred[S],
          inv: pred[S],
          pre: [I->pred[S]]
       ]: THEORY

BEGIN

ASSUMING
  i: VAR I
  x,y: VAR S

  init: ASSUMPTION  EXISTS x: init(x) & inv(x)

  no_deadlock: ASSUMPTION
     pre(i)(x) & inv(x) => EXISTS y: inv(y) & Imap(i)(x,y)

ENDASSUMING

Reachable(x):INDUCTIVE bool =
   init(x) & inv(x)
OR EXISTS y,i : Reachable(y) & pre(i)(y) & inv(x) & Imap(i)(y,x)

Invariant_holds: LEMMA  FORALL x: Reachable(x) => inv(x)

END machine
```

A theory starts with a name, followed by a number of parameters (types or constants), then the keyword THEORY. *The specification itself is between* BEGIN *and* END name. *The*

parameters may be dependently typed (the types `I`, `O` *and* `S` *are used in the other parameters). A theory can be imported by another theory, which can provide actual types and values for the parameters.*

`ASSUMPTION`*s go into the* `ASSUMING` *section. When the theory is imported with actual parameters, a proof obligation is generated for the importing theory, that the assumptions hold on the instance (see section 3.5.2). In the imported theory, the assumptions can be used just as if they were* `AXIOM`*s.*

We also use an `INDUCTIVE` *definition. This means that we may use the function being defined, i.e.* `Reachable`*, in the right hand side of the definition. Thus the set of reachable states is defined as the smallest set of states containing the initial state, and closed under transitions caused by input events that satisfy the precondition.*

Such inductive definitions automatically generate a powerful induction scheme, which reduces the task to prove P for all reachable states, to the task of proving P for the initial state, and proving that for each possible input event, P is maintained by the corresponding state transition.

3.5.2 Checking the verification proof obligations

We can now proceed with importing the theory of the previous section:

```
IMPORTING machine[IEvents,OEvents,State,
                  Input_table,Output_table,
                  initial,Invariant,Precondition]
```

This will, on type checking, automatically generate two proof obligations. The first one is to check that the initial state satisfies the invariants. It reads as follows:

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

In the interactive prover, this lemma can be proved by first instantiating `x` to the state below, where all sets are empty. Then the single PVS command `(GRIND)` finishes the proof by automatically verifying that all constraints are satisfied. The complete proof can be found in Appendix D.2

```
(# sensor_ids  := emptyset,
   system_ids  := emptyset,
   joined      := emptyset,
   system_info := LAMBDA tn : epsilon! t : TRUE
#)
```

Elements of record types can be constructed with `(# ... #)`*, with assignments to all record labels. An alternative way of defining a record is by modifying an existing one, we could e.g. write* `X WITH [sensor_ids := S]`*. In the new record, the omitted fields are taken from* `X`*, and the* `sensor_ids` *field is set to* `S`*.*

In order to find an arbitrary track state, we used `epsilon!`*, Hilbert's choice operator.* `epsilon! (x:T): P` *denotes an arbitrary element* `x` *from* `T`*, for which* `P(x)` *holds. This is type correct for nonempty types. We used* `TYPE+`*, so the set of system track states is nonempty.*

The second proof obligation is displayed below. It states roughly speaking that transitions are possible, whenever the environment guarantees the preconditions. We need some auxiliary lemmas, viz. that for each individual input event a transition is possible. We give

an indication of the main steps in the proofs. Full proofs can be found in appendix D.2. Note that it is not needed to type the full proof at once; a proof is constructed interactively with PVS. The process of finding a proof is beyond the scope of this report. (In fact the auxiliary lemmas go above the `IMPORTING` clause in the actual PVS file.)

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

A `LEMMA` *declaration can be compared to an* `AXIOM` *declaration. The only difference is that a lemma requires a proof. (Axioms don't appear in the proof status reports generated by PVS, see appendix C.) As with axioms, PVS takes the universal closure of each lemma, in this case* `FORALL X,sn,s`.

Here is an outline of the proof: By the axiom `corex` (3.3.1) there exists a system track state `t`, such that `s` and `t` correlate. By the precondition there exists a fresh system track identifier `tn`. Then we can instantiate `Y` with the following term:

```
(# sensor_ids := add(sn,sensor_ids(X)),
   system_ids := add(tn,system_ids(X)),
   joined := add((sn,tn),joined(X)),
   system_info := system_info(X) WITH [tn:= t] #)
```

It remains to verify that the invariant and the `new_sensor_track` relation hold. This can be done nearly automatically, using a number of variants of `(GRIND)`.

Next we consider the `wipe` event:

```
wipe_next: LEMMA
  Invariant(X) & Precondition(wipe(sn))(X)
=> EXISTS Y: Invariant(Y) & wipe_sensor_track(sn)(X,Y)
```

The proof runs as follows: Let `tn` be the system track joined to `sn` in `X`. Then `Y` can be found by removing `sn` from `sensor_ids(X)` and the pair `(sn,tn)` from `joined(X)`. In order to satisfy `constraint3`, we also have to remove `tn`, in case `sn` is the only sensor track joined to the tactical track `tn`.

Next we check the `update` event:

```
update_next: LEMMA
  Invariant(X) & Precondition(update(sn,s))(X)
=> EXISTS Y: Invariant(Y) & update_sensor_track(sn,s)(X,Y)
```

This can be proved very simply: by applying lemma `wipe_next` we obtain `Z`, satisfying the invariant, where `sn` has been removed. By lemma `new_next` we obtain `Y`, again satisfying the invariant, and incorporating the pair `(sn,s)`. In proving all the side conditions (which PVS forces you to do quite painfully), we need axiom `cordecor` (3.3.1).

All these pieces can be glued together to prove the second verification proof obligation, which expresses deadlock freedom in any environment satisfying the precondition. The proof proceeds by case distinction over the possible input events, each case using one of the previous lemmata.

```
IMPORTING1_TCC2: OBLIGATION
  (FORALL (i: IEvents, x: State):
     Precondition(i)(x) & Invariant(x)
        => (EXISTS (y: State): Invariant(y) & Input_table(i)(x, y)));
```

Having this, we can request an automatically generated status report from PVS, in order to check that all proofs have been carried out. The result can be found in appendix C, which contains the proof summary for theory `spec1`.

Remarks

The proofs in this section were non-trivial, because we had to construct the next state `Y`, satisfying all requirements. However, the actual construction of the next state is the main concern that must be addressed during the implementation of the system. Moreover, in view of the refinements of Chapter 4, the proof obligations seem superfluous: the final correct implementation is the ultimate proof that the requirements are consistent.

However, notice that we want to establish consistency *before* addressing an implementation. The construction used in the proof might be much simpler, as we only have to prove *existence* of *some* next state, while the designer also has to take issues of efficiency and resources into account. Actually, in our previous proof, an update is "implemented" by a wipe followed by an initiate, which is much simpler than the algorithm used in actual implementations.

3.5.3 Validation

Validation by means of formal challenges is deferred to the refined specification in section 4.2.3, where more requirements are present. To give an indication of what could be proved in this abstract specification, consider the following exercises:

Exercise 1. (beginner) Prove that the invariant forces that in the initial states also the set of system tracks is empty.

Exercise 2. (experienced) Prove that for all reachable states, the set of sensor tracks is finite.

Exercise 3. (expert) Prove that for all reachable states, the set of system tracks is finite.

Chapter 4

Refinements

This chapter introduces two refinements of the top level specification developed in the previous chapter. The first refinement adds more requirements to the state transitions; the effect is that the specification becomes more operational. The second refinement adds the detailed computations on the track states, so the specification becomes more detailed.

The refinement relation between the top specification and the first refinement is rather informal, although we think that the relation could be formalized. The relation involves a more structured data specification, and it reduces non-determinism.

In order to show that the second refinement formally refines the first, we parameterize the first refinement. The parameters are instantiated by the second refinement. So the notion of refinement is a logical one: All models permitted by the second refinement, are a model of the first refinement.

4.1 First refinement: operational requirements

We refer to the specification of Chapter 3 as the top specification. The first refinement incorporates three major changes compared to the top specification:

- Operational requirements are added.
- The invariant is converted into subtypes.
- The specification is parameterized.

The top specification was rather global. We left out the kinematic computations, and the connection between sensor and system tracks was specified in a number of invariants. However, the specification allows a number of unintended implementations. So the main purpose of this first refinement is to add extra requirements.

The top specification allows for instance that each system track is joined to only one sensor track; so the sensor tracks are never really joined. The top specification also allows that the join relation is completely changed after each new and update event, which would result in a very unstable picture for the operator.

These problems are removed in the first refinement, by stating more exactly *how* the next state is computed from the previous one. So we add requirements to the state transitions (section 4.1.3), stating how sensor tracks shall be joined to system tracks, and how the system track state shall be updated. As a side effect, certain properties are automatically

induced by the strengthened state transitions, and need not be required as an invariant. Another effect is that the specification becomes more operational.

We also shift some constraints from the invariant to the types of the variables. This also makes the specification more operational, because assertions (constraints) are converted into structural properties of the model (construction). This is explained in section 4.1.2

Finally, the top specification contained a number of uninterpreted type parameters and constants, and axioms to restrict their interpretations. We now use the parameter mechanism and assumptions on the parameters, thus avoiding axioms, and making formal refinements possible (section 4.1.1).

4.1.1 Parameterizing the specification

In the top specification, some types and constants have not been defined. They were left uninterpreted, thus allowing some degree of freedom in the implementation. In order to be able to refine the specification, we now actually specify these uninterpreted symbols as parameters of the theory. In fact, what we get is a set of specifications, one for each value of the parameters. A logical refinement can be seen as a partial instantiation of these parameters, thus decreasing the set of possible specifications. In PVS this corresponds to importing the theory with actual parameters.

The set of identifiers for sensor and system tracks, and the actual contents of their states are type parameters. The correlation and decorrelation criteria are constant parameters. We also introduce the parameters `initiate`, which starts a system track on a given sensor track, and `update`, which updates a system track with a given sensor track.

```
spec2
 [ Sensor_track, Sensor_track_state,
   System_track, System_track_state: TYPE+,
   correlates, decorrelates:[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 spec2
```

We need some assumptions on these parameters, which correspond with the axioms of section 3.3.1. The correlation and decorrelation criteria should be exclusive (cf. axiom `cordecor`), a sensor track and the system track initiated on it should correlate (this replaces axiom `corex`), and the decorrelation test on an updated system track should not fail, unless it failed already before the update. In section 4.3.4 this theory is imported by the second refinement, and the assumptions have to be shown for the actual parameters.

```
ASSUMING

s: VAR Sensor_track_state
t: VAR System_track_state

cordecor: ASSUMPTION  NOT (correlates(s,t) & decorrelates(s,t))
corin:    ASSUMPTION  correlates(s,initiate(s))
decorup:  ASSUMPTION  decorrelates(s,update(s,t)) => decorrelates(s,t)

ENDASSUMING
```

The set of input and output events is unchanged. Therefore we now proceed with the information model. The reader is referred to Section B.2 for the complete specification.

4.1.2 Information model: subtypes vs. invariants

We have made a distinction between the *structure* of a state, defined as a record with a number of typed fields, and the *invariant* properties of a state, consisting of a conjunction of predicates on this record. However, due to the expressiveness of PVS as it comes to type definitions, this border is not clear. In principle, every specification can do without separate invariants. This strength is due to the combination of *subtyping* and *dependent typing* in PVS.

In section 3.2, `system_info` is defined as a total function with domain `System_track`. However, using subtypes we can restrict its domain to the system tracks actually in `system_ids`.

In section 3.2.2, the joined relation is restricted to pairs in the current sensor and system track sets by `constraint1`. Moreover, `constraint2` requires the joined relation to be functional and total. We can avoid these constraints by giving `joined` a function type, instead of a relation type. We then get the following state definition:

```
State: TYPE =
  [# sensor_ids: setof[Sensor_track],
     system_ids: setof[System_track],
     system_info: [(system_ids)->System_track_state],
     joined: [(sensor_ids)->(system_ids)]
  #]
```

Consider a set `X:setof[t]`. *Recall that in PVS this is equivalent to* `X:[t->bool]`. *We can now form a subtype of type* `t`: {`x:t | X(t)`} *(set comprehension), consisting of precisely those elements of type* `t` *that are in* `X`. *This type can be abbreviated as* `(X)`.

The example uses subtypes, but also dependent typing, because the type of the third field depends on the value of the first two fields. In this case, the domain of `system_info` *is restricted to the sensor tracks actually in* `sensor_ids`, *the only tracks of which the state vector is known to the system. This refined definition is much more precise than the original one in Section 3.2.1.*

We now declare a number of variables:

```
tn: VAR System_track
sn: VAR Sensor_track
X,Y: VAR State
ie: IEvents
oe: OEvents
```

We don't have constraints in the refined specification. As explained before, `constraint1` and `constraint2` are superfluous. The definition of the state transitions (section 4.1.3) subsume `constraint3`. These statements are formally proved in section 4.2. So we have the empty invariant:

```
Invariant(X):bool = TRUE
```

The initial state can be defined straightforwardly. The sets of sensor and system tracks are defined to be empty. Note that both functions in the state are determined by the fact that their domain is empty.

```
initial(X):bool = empty?(sensor_ids(X)) & empty?(system_ids(X))
```

Intermezzo: subtypes and TCCs

The introduction of subtypes above has some implications. Each time a term of a subtype is used, it must be checked that the subtype restrictions indeed hold. As this may be undecidable, the PVS type checker generates the corresponding subtype TCCs (type check correctness conditions). These TCCs have to be proved interactively using the PVS proof checker. A theory is only regarded type correct if all generated TCCs have been proved.

As an example, consider the following constraint, which is `constraint3` adapted to the new type `State`. We don't put boxes around this PVS text, because it doesn't belong to the specification; it is only there for expository purposes:

```
constraint3??(X):bool = FORALL tn:
  system_ids(X)(tn) => EXISTS sn: tn = joined(X)(sn)
```

This appears to be not type correct! This is because the `joined` function is applied to an arbitrary sensor track, and it is defined only for sensor tracks in `sensor_ids`. This can be repaired as follows:

```
constraint3(X):bool = FORALL tn: system_ids(X)(tn) =>
  EXISTS sn: sensor_ids(X)(sn) & tn = joined(X)(sn)
```

Although these extra checks and conditions seem a nuisance, it helps in finding errors in the specification. Consider the following unintended alternative formulation of the third constraint:

```
Problematic_constraint3(X):bool =
  FORALL tn: EXISTS sn: joined(X)(sn,tn)
```

This is unintended, because it requires that *any conceivable* system track (not only those known to the system) is joined to some sensor track. This error would go undetected without subtyping, but in the context of the state definition above, this would result in an unprovable TCC. This error would also have been detected using the method of section 2.3.2.

Summarizing, subtypes can be used to shift requirements from the invariants to the types of the state variables. The result is the generation of more TCCs. This yields more work, but it may reveal some errors.

4.1.3 Refining state transitions

A naive implementation of `new_sensor_track` in section 3.3.1 can easily decide always to create a new system track. This is ruled out in the current refinement, by requiring that creating a new system track is only allowed if all correlation tests with existing system tracks fail. Note that this requires "optimal performance" of the system, whereas the original specification would allow a certain degree of degradation. Similarly, we require that when updating a sensor track, disjoining is only possible if there is a decorrelation. Furthermore, tracks not involved in the operation shall remain unchanged.

Sensor track initiation. Hence we introduce a predicate `no_correlation`, which states that there is no correlation between a given sensor track and any of the current system tracks.

```
no_correlation(s,X):bool =
  FORALL tn: system_ids(X)(tn)
             => NOT correlates(s,system_info(X)(tn))
```

The type of `s` *was declared in the* `ASSUMING` *section on page 30.*

If a new sensor track is received, a new system track is initiated, unless it is possible to join the new sensor track to an existing system track. To this end, two auxiliary state transitions are introduced: `new_system_track` and `correlate_to_system_track`.

Let a new sensor track `sn` with state `s` be given. Assume that it correlates with some system track `tn`. Then `sn` should be added to the set of sensor tracks, the pair `(sn,tn)` should be added to the join relation, while the set of system tracks remains unchanged. The system track state is to be recomputed with the auxiliary function `update`, which was declared before. It might also be the case that more than one system track correlates with `sn`, in which case the result is not completely determined.

Note that `correlate_to_system_track` is not applicable if there is no system track with which `sn` correlates.

```
correlate_to_system_track(sn,s)(X,Y):bool =
  EXISTS tn:
    system_ids(X)(tn)
  & correlates(s,system_info(X)(tn))
  & Y = X WITH
    [sensor_ids := add(sn,sensor_ids(X)),
     joined := joined(X) WITH [sn:=tn],
     system_info := system_info(X)
        WITH [tn:=update(s,system_info(X)(tn))]]
```

We use the `WITH` *construction on records, to obtain* `Y` *from* `X`*. Note that the set* `system_ids` *is not changed. The* `WITH` *construction can also be used on functions. The function* `f WITH [x := n]` *denotes the function that equals* `f` *on all arguments apart from* `x`*, where it returns* `n`*. This construct is used to change the* `joined` *and* `system_info` *function.*

The alternative transition, which creates a new system track, is defined below. This is again non-deterministic, because any identifier `tn` is allowed, as long as it is a fresh identifier. The auxiliary function `initiate` is used to compute the initial system track state.

```
new_system_track(sn,s)(X,Y):bool =
  no_correlation(s,X)
& EXISTS tn:
    (NOT system_ids(X)(tn))
  & Y =
    (# sensor_ids := add(sn,sensor_ids(X)),
       system_ids := add(tn,system_ids(X)),
       joined := joined(X) WITH [sn:=tn],
       system_info := system_info(X) WITH [tn := initiate(s)] #)
```

Finally, the `new_sensor_track`-transition of section 3.3.1 can be refined as follows:

```
new_sensor_track(sn,s)(X,Y):bool =
   new_system_track(sn,s)(X,Y)
OR correlate_to_system_track(sn,s)(X,Y)
```

Sensor track deletion. Wiping a sensor track involves deleting it from `sensor_ids`. The joined relation has to be restricted accordingly. If the corresponding system track was joined to the wiped sensor track only, it will become unsupported by any measurement, and it should be removed. To this end, we use the auxiliary function `filter`, which removes unrelated system tracks.

```
filter(X):State =
  X WITH
    [system_ids := {tn | system_ids(X)(tn)
       & EXISTS sn: sensor_ids(X)(sn) & tn=joined(X)(sn)},
     system_info := restrict(system_info(X))]
```

The notation `{x | S}` *is not only used for subtyping, but also to define a subset by set comprehension. We use the function* `restrict` *from the library, which restricts a function to a smaller domain. The actual domain is derived by PVS from the context.*

```
wipe_sensor_track(sn)(X,Y):bool =
  sensor_ids(X)(sn)
& Y = filter(X WITH [
   sensor_ids := remove(sn,sensor_ids(X)),
   joined := restrict(joined(X))])
```

Sensor track updates. When a sensor track update is received, the state of the assigned system track shall be updated accordingly unless the decorrelation test succeeds. In the latter case, the sensor track shall be assigned to another system track (existing or newly created). To this end we can use `new_sensor_track`. The original system track may become unrelated, so we `filter` the intermediate result.

```
update_sensor_track(sn,s)(X,Y):bool =
  sensor_ids(X)(sn)
& LET tn = joined(X)(sn), t = system_info(X)(tn) IN
  IF decorrelates(s,t)
  THEN EXISTS (Z:State): new_sensor_track(sn,s)(X,Z) & Y=filter(Z)
  ELSE Y = X WITH
           [system_info:= system_info(X) WITH [tn := update(s,t)]]
  ENDIF
```

We use the LET ... IN ... *construction as an abbreviation mechanism. We also use the* IF ...THEN ...ELSE ...ENDIF *construction of PVS. Note that the intermediate result* Z *is only for internal use, so it is encapsulated with existential quantification.*

4.1.4 Triggers

The triggers remain nearly unchanged w.r.t. the top specification (section 3.3.2). Only the references to `joined` in `detect_decorrelation` have to be modified. We refer to appendix B.2 for the complete specification. Here we only incorporate the new definition of `detect_decorrelation`:

```
detect_decorrelation(sn)(X,Y):bool =
  sensor_ids(X)(sn) & sensor_ids(Y)(sn)
& joined(X)(sn) /= joined(Y)(sn)
```

This concludes the specification of the first refinement. The input and output tables, as well as the preconditions, remain unchanged. The reader is referred to section B.2 for the complete specification.

4.2 Analysis of first refinement

4.2.1 Parsing and type checking

Due to the subtyping, PVS generates a number of proof obligations, called TCCs. These have to be proved using the PVS prover, because in general TCCs are not decidable. Only after these proofs are completed, the specification can be regarded as type correct. In this case, PVS generates eight type check conditions (besides the verification proof obligations), which PVS is able to prove automatically. One of these obligations is:

```
correlate_to_system_track_TCC1: OBLIGATION
      (FORALL (X: State, s: Sensor_track_state, sn: Sensor_track, tn):
         system_ids(X)(tn) AND correlates(s, system_info(X)(tn))
             IMPLIES add(sn, sensor_ids(X))(sn));
```

This rather trivial TCC is generated because in `correlate_to_system_track`, we add `sn` to the domain of `joined`. By the type of `joined`, `sn` should be in the new `sensor_ids`. Note that the typing rules are context dependent, so in the proof we may use for instance that `tn` is a known system track. In this case, the context is not needed, however.

4.2.2 Verification of preconditions

Next, analogously to section 3.5.2, we prove for the individual state transitions, that if their precondition holds then there exists a successor state. These lemmas are a preparation for the verification proof obligation. The proofs are similar to those of the previous section (in fact finding the next state is more straightforward, due to the operational description).

```
e1: LEMMA system_ids(X)(tn) & correlates(s,system_info(X)(tn)) =>
     EXISTS Y: correlate_to_system_track(sn,s)(X,Y)

e2: LEMMA no_correlation(s,X) & (EXISTS tn: NOT system_ids(X)(tn))
    => EXISTS Y: new_system_track(sn,s)(X,Y)

e3: LEMMA (EXISTS tn: NOT system_ids(X)(tn))=>
           EXISTS Y: new_sensor_track(sn,s)(X,Y)

e4: LEMMA sensor_ids(X)(sn) &  (EXISTS tn: NOT system_ids(X)(tn))
    => EXISTS Y: update_sensor_track(sn,s)(X,Y)

e5: LEMMA sensor_ids(X)(sn)
    => EXISTS Y: wipe_sensor_track(sn)(X,Y)
```

Next we import the generic machine model (the same as before), which automatically generates the verification proof obligations.

```
IMPORTING machine[IEvents,OEvents,State,
                  Input_table,Output_table,
                  initial,Invariant,Precondition]
```

This generates the verification proof obligations. The first of them is proved by providing a possible initial state. The other one is more complicated and is displayed below. The proof is by induction over the set of input actions and uses the lemmas `e3`–`e5` for the subcases.

```
IMPORTING1_TCC2: OBLIGATION
     (FORALL (i: IEvents, x: State):
        Precondition(i)(x) & TRUE
            => (EXISTS (y: State): TRUE & Input_table(i)(x, y)));
```

4.2.3 Validation by formal challenges

We now prove some extra theorems, which we expect to hold. Most of these theorems are inspired by the top level specification in Chapter 3, so they give some confidence in the fact that this really is a refinement.

Proving invariants

We suggested that by adding subtypes to the state definition, `constraint1` and `constraint2` would become implicit by the specification. Let us check whether this is indeed the case:

```
constraint1: LEMMA
   sensor_ids(X)(sn) & tn = joined(X)(sn) => system_ids(X)(tn)

constraint2: LEMMA
  sensor_ids(X)(sn) => exists1! tn: tn = joined(X)(sn)
```

Both can be proved with a single `GRIND` command. This is not surprising in view of the type of the `joined` function.

Proving new invariants

Note that `constraint3` cannot be proved as easily as `constraint1` and `constraint2`, because it doesn't follow from the information model. But all state transitions are defined in such a way that `constraint3` holds for all reachable states. Let us first (re)define `constraint3`:

```
constraint3(X):bool = FORALL tn:
  system_ids(X)(tn) =>
    EXISTS sn: sensor_ids(X)(sn) & tn = joined(X)(sn)
```

Next, we prove that each state transition maintains `constraint3`:

```
l1: LEMMA constraint3(X) & new_system_track(sn,s)(X,Y)
          & (NOT sensor_ids(X)(sn)) => constraint3(Y)

l2: LEMMA constraint3(X) & correlate_to_system_track(sn,s)(X,Y)
          & (NOT sensor_ids(X)(sn)) => constraint3(Y)

l3: LEMMA constraint3(filter(X))

l4: LEMMA constraint3(X) & new_sensor_track(sn,s)(X,Y)
          & (NOT sensor_ids(X)(sn)) => constraint3(Y)

l5: LEMMA constraint3(X) & update_sensor_track(sn,s)(X,Y)
          => constraint3(Y)

l6: LEMMA constraint3(X) & wipe_sensor_track(sn)(X,Y)
          => constraint3(Y)
```

As can be seen from these lemmas, certain assumptions on the environment have been made, in order to prove that constraint3 is a global invariant. These assumptions all follow from the precondition. So we have the following invariant:

```
constraint3: LEMMA Reachable(X) => constraint3(X)
```

This is proved by induction over the reachable states `(RULE-INDUCT "Reachable")`, where the lemmata before (`l4`–`l6`) serve to prove the induction step.

Updating never leads to decorrelation

In the top specification, it is required (3.3.1) that a sensor track and the system track joined to it don't decorrelate after a sensor track update. We might want to check this in the

refinement:

```
update_no_decorrelation: LEMMA
  update_sensor_track(sn,s)(X,Y) =>
    NOT decorrelates(s,system_info(Y)(joined(Y)(sn)))
```

The proof of this lemma reads as follows:

```
 (GRIND :IF-MATCH NIL)
 (("1" (USE "corin") (USE "cordecor") (PROP))
  ("2" (USE "cordecor") (USE "decorup") (PROP))
  ("3" (USE "decorup") (PROP)))
```

Here GRIND is an automatic tactic, which among others unfolds all definitions in the proofs. The tactic USE allows us to use previously proved theorems, axioms or assumptions. Finally PROP solves the goal using propositional logic only. This exercise learns us that under reasonable assumptions (and we know *exactly* which) a sensor track update doesn't introduce decorrelation.

Causal relationships between events

The link between input and output events is quite indirect in the specification. Therefore, it is interesting to inspect what output events may be caused by the input events, and what output events can occur simultaneously. We first define these concepts (this could be done in the abstract machine model):

```
oe1,oe2: VAR OEvents
OE: VAR setof[OEvents]
tn1,tn2: VAR System_track
t1,t2: VAR System_track_state
sn1: VAR Sensor_track

May_cause(ie,OE):bool = EXISTS X,Y:
    Precondition(ie)(X)
  & Input_table(ie)(X,Y)
  & forall oe: OE(oe) => Output_table(oe)(X,Y)

May_appear_together(OE):bool =
  EXISTS ie: May_cause(ie,OE)
```

Next we can prove the following interesting theorems (all by just typing GRIND):

```
CONVERSION singleton[OEvents]

L1: LEMMA NOT May_cause(new(sn,s), wipe(tn))
L2: LEMMA NOT May_cause(wipe(sn), new(tn,t))
L3: LEMMA NOT May_cause(wipe(sn), update(tn,t))
L4: LEMMA NOT May_cause(wipe(sn), warn(sn1))
L5: LEMMA NOT May_appear_together( (: new(tn1,t1),update(tn2,t2) :))
```

Above we wrote a single output action, instead of a set of output actions. This is allowed,

because we introduced a `CONVERSION`*, viz. the* `singleton` *function from the prelude, instantiated to output events. There is also a hidden conversion from lists to sets, as we used the list notation* `(: x,y,z :)` *in the final lemma.*

So one of the conclusions is that initiating a new sensor track can never lead to wiping an existing system track (`L1`). Furthermore, wiping a sensor track cannot lead to the update or the initiation of any system track (`L2, L3`), at most to a wipe (it might go undetected). Finally, it is never the case that in one transition some system track is created, and another updated (`L4`).

4.3 Logical refinement: adding detail

In this section, we add some details to the specification. This is achieved by specifying a number of parameters. A parameterized theory can be seen as a set of possible specifications. By providing actual parameters, we obtain a subset of the set of possible specifications. This can be seen as a refinement, because it reduces the implementation freedom by incorporating additional decisions.

In this case, we specify the actual track states, and the kinematic computations. Formally, the specification of section 4.1 is imported, and a number of parameters is instantiated. This amounts to the following modifications:

- It is decided that the track states consist of three dimensional position and velocity vectors, and identification information. Moreover, sensor tracks contain information on their source.
- The correlation and decorrelation criteria are now defined. The criteria involve that the distance is within a certain margins and that identifications are compatible.
- The functions initiate and update are defined.

For presentation purposes the actual computations have been simplified. In reality, we would formalize e.g. Kalman filtering here. There is no fundamental problem in formalizing this filtering process, but it would take some time. The current formalization is sufficient to show how such details can be added to a specification, and how PVS supports specifications with computations on real numbers. The simplifications that we have made are:

- The accuracy of measurements and tracks is omitted.
- We work in Cartesian coordinates only.
- Time alignment is skipped.

Because accuracies are dropped, the only thing we can do when updating a system track with a sensor track, is copying the latest sensor track.

4.3.1 Specification of track states

The second refined theory still has parameters for the track identifiers (these are mere implementation details), for the `Source` of the tracks, and for the `Margins` that play a role in the correlation and decorrelation criterion. So it would be possible to refine this theory even further.

```
spec3
 [ Sensor_track, System_track,
   Source: TYPE+,
   Margin1,Margin2: posreal
 ]: THEORY
BEGIN

ASSUMING
  Margins: ASSUMPTION Margin1 < Margin2
ENDASSUMING

..... (see below)

END spec3
```

A track state consists of identification and kinematic information. Only finitely many identifications are distinguished:

```
Identification: TYPE = {friend, hostile, pending}
```

Here we used enumeration types, which are a special case of abstract data types. The elements of enumeration types are pairwise distinct, and enumerate the whole type.

The kinematic information of a track contains its (x, y, z)-position and velocity, as real numbers.

```
Kinetic: TYPE = [# px,py,pz,vx,vy,vz: real #]
```

`real` *is the built-in type of real numbers.*

We are now in a position to define the state variables of a single sensor track and system track:

```
Sensor_track_state: TYPE =
[# source: Source,
   identification: Identification,
   kinetics: Kinetic
#]

System_track_state: TYPE =
[# identification: Identification,
   kinetics: Kinetic
#]
```

Note that records can be nested. Recall that the field names can be used as accessors. Given a system track state `t`*, we can find its velocity in the* x*-coordinate by the term* `vx(kinetics(t))`.

4.3.2 Kinematic computations

We continue with the kinematic computations. At initiation, the state of the new system track is just copied from the sensor track. For updating a track, we just copy the sensor track. In reality, we would take the statistic mean of the system and sensor track, taking

the accuracies into account.

```
s: VAR Sensor_track_state
t: VAR System_track_state

initiate(s):System_track_state =
  (# identification := identification(s), kinetics:=kinetics(s) #)

update(s,t):System_track_state =
  (# identification := identification(s), kinetics:=kinetics(s) #)
```

4.3.3 Correlation criteria

There are two correlation criteria. The distance criterion is straightforward. In reality, a statistic distance would be used. The criterion on identities is that they are not conflicting, which is specified by a table.

```
distance(s,t):real =
  LET k1 = kinetics(s), k2 = kinetics(t) IN
    (px(k1)-px(k2)) * (px(k1)-px(k2))
   +(py(k1)-py(k2)) * (py(k1)-py(k2))
   +(pz(k1)-pz(k2)) * (pz(k1)-pz(k2))
```

We can use ordinary arithmetic operators on real numbers, like `+, -, *, /`*. Natural numbers (*`nat`*) are a subtype of the real numbers. They can be typed as* `0,1,2,...`*. There is no floating point notation for real numbers; only rational numbers can be written down. The existence of numbers (like* $\pi, \sqrt{2}$*) can be proved, but there is no notation for them.*

```
conflicting(s,t):bool =
  LET id1 = identification(s), id2 = identification(t) IN
TABLE
  id1, id2 |[pending| friend |hostile]|
%--------------------------------------
|pending   | FALSE  |  FALSE | FALSE ||
|friend    | FALSE  |  FALSE | TRUE  ||
|hostile   | FALSE  |  TRUE  | FALSE ||
ENDTABLE%-----------------------------
```

PVS supports a tabular notation, which can be used for unary and binary functions. Tables can also be nested. The number of terms after the opening `TABLE` *keyword defines the dimension of the table. The rest of the format is self-explanatory. Note that anything following a* `%` *is a comment only.*

```
correlates(s,t):bool =
    distance(s,t) <= Margin1 * Margin1
  & NOT conflicting(s,t)

decorrelates(s,t):bool =
     distance(s,t) >= Margin2 * Margin2
  OR conflicting(s,t)
```

This completes the specification of the track states and the computations.

4.3.4 Analysis of the second refinement

In order to check that the second refinement actually refines the first one, we can now import the first refinement, with as actual parameters the definitions from the previous sections.

```
IMPORTING spec2
  [Sensor_track,Sensor_track_state,
   System_track,System_track_state,
   correlates,decorrelates,initiate,update]
```

This automatically generates a number of proof obligations, corresponding to the assumptions made in the first refinement (section 4.1.1) These proof obligations read as follows:

```
IMPORTING1_TCC1: OBLIGATION
      (FORALL (s: Sensor_track_state, t: System_track_state):
         NOT (correlates(s, t) & decorrelates(s, t)));

IMPORTING1_TCC2: OBLIGATION
      (FORALL (s: Sensor_track_state): correlates(s, initiate(s)));

IMPORTING1_TCC3: OBLIGATION
      (FORALL (s: Sensor_track_state, t: System_track_state):
         decorrelates(s, update(s, t)) => decorrelates(s, t));
```

These obligations can be proved by `GRIND` except the first one, which needs the assumption that we have made: `Margin1 < Margin2`. Having proved these obligations, we know that the second refinement is a logical refinement of the first, in the sense that it admits less implementations.

Because the second refinement `spec3` is an instance of the first refinement `spec2`, every theorem proved in the first refinement automatically holds in the second. This holds especially for the verification proof obligations, so a further verification is not needed.

Chapter 5

Conclusion

5.1 Possibilities for future work

5.1.1 Extension to the specification method

It is expected that the method has to be extended. We think that more case studies are needed to decide on these extensions. Some simple extensions are:

- Addition of internal events (or transitions without input events)
- Triggers that depend on input events (this can already be encoded by storing the last event in the state).
- Invent more proof obligations, for instance to verify that each trigger will sometimes be true.

The reason that these extensions have not been incorporated is first of all pragmatic: they were not needed for the examples considered. Moreover, we think that especially the addition of internal events could, when used carelessly, compromise the abstractness of the specification. This option becomes interesting if the same method will be used for describing the design of the system.

We now describe other extensions, which we regard useful.

Modularity

In order to get more structured specifications, a specification could be decomposed into logical components. These don't necessarily coincide with the physical components of the system. The physical decomposition 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 components is consistent
- to construct a particular view on the whole system.

With the latter requirement we mean that it should be simple to find the invariant of the whole system, the state of the whole system, the transitions of the whole system, etc.

In [PHJ99], we proposed a modular interpretation of temporal logic, which satisfies both requirements above. This solution has not yet been fully integrated into the method. Future work on modularity could start with this integration. Temporal logic is also mentioned in the next section.

Protocol on occurrence and timing of events

The physical interpretation of the Mealy machine in this paper has two extremes:

- The output events on a state transition should occur before the next input event.
- The output events on a state transition should occur eventually.

The first interpretation is too strict for distributed systems, where input events can happen on different locations. The second interpretation allows too much freedom, because the reaction of the system can be postponed arbitrarily long.

An extra specification block could specify requirements on the actual occurrence of these output events. To this end one can use either process algebra, which has an operational flavour, or temporal logic. In the latter formalism, one can express for instance that a certain output event shall always occur before a particular input event happens. Also real time constraints can be expressed in temporal logic, for instance that some output event shall happen within t seconds after a certain input event.

In [PHJ99] we proposed the use of temporal logic. A general problem is that it is hard to directly specify a system in temporal logic. Future research should make clear which requirements are best expressed in temporal logic. The specification patterns that will emerge could be captured by a number of "template" formulae. For certain requirements, for instance the life cycle of a track, a specification in process algebra might be simpler.

5.1.2 Semantics

Not all semantical issues have been solved. We think that these issues can only be resolved after some experiments with the design process. Also typical issues of the underlying architecture, for instance SPLICE, will emerge here. At this moment it is too early to take any decisions here. We discuss three semantical issues in more detail: equivalence of specifications (this is related to what is observable), refinement of specifications, and the composition of machines.

Equivalence

A requirements specification is mapped to a particular machine model. We can ask the question which aspects of the machine are observable. As said before, the input and output events form the interface with the environment. However, in the Mealy machine, also the structure of the internal state is visible. This problem can be solved by defining a suitable equivalence relation on machines, in such a way that two machines are equivalent if they cannot be distinguished by the observations. This question has not been addressed in our project. The literature gives a wide variety of equivalence relations. Some well-known possibilities are (in order of decreasing granularity):

1. Two machines are equivalent if they are exactly the same.
2. Two machines are equivalent if they are isomorphic (i.e.: there is an isomorphism between states, maintaining all transitions). This means that the structure of the states is discarded.
3. Two machines are equivalent if they are strongly bisimilar.
4. Two machines are equivalent if they produce the same input-output traces of the form $i_1, O_1, i_2, O_2, \ldots$, where i_k is the k-th input event, and O_k the set of triggered output events.

5. Two machines are equivalent if they produce the same function of input to output, that is $(i_1, \ldots, i_n) \mapsto (O_1, \ldots, O_n)$.

We feel that the minimal identification should be (2) because the state should be internal. The last definition is used in language theory, where Mealy machines were first studied; this is not well-suited for system specifications, as it neglects the causal relationship between input and output events in the reactive behaviour. Option (4) is also often used. In fact this option was chosen in [PHJ99], as it is the basis of linear temporal logic. However, using traces, the precise interaction of the machine with an operator cannot be modeled, because choice points aren't preserved under trace equivalence. The precise effect of the combination of traces and input enabledness has still to be studied. Probably the best choice is to adapt bisimulation to an asynchronous setting, where events that are independent (e.g. physically distributed) can be swapped.

Refinement

It is important to define when a system can be seen as an implementation of the requirements specification. The implementation needs not be equivalent, but merely refines the specification. A refinement may involve additional choices, which can be recognized already in our refinements at several places:

1. Data refinement: choosing more concrete data, such as lists instead of sets, functions instead of relations, etc.

2. Reduction of non-determinism. If the specification allows two different behaviours on a certain input, the implementation may choose one of them.

3. Addition of internal actions. A state transition can be refined into several smaller steps. Care has to be taken that the environment cannot view an intermediate state; on the interface the invariant has to be kept.

4. Invariants can be eliminated by "implementing" them on each transition.

Several of these steps would result in the same underlying machine, which is only specified more operationally, viz. 4 and 1. Data refinement (1) might be visible if the data appears as parameters of the events. Other steps, for instance 2 and 3, constitute a modification of the underlying system. Future research should solve which refinement steps are necessary, and how they can be formally defined. Maybe the literature on forward and backward simulations is applicable here.

Composition

The final semantical issue is composition of machines. This is important in system integration, and in the research for modularity. It would be nice if two Mealy machines could be composed, e.g. by communication: one of them could consume output events of the other. As the second machine belongs to the environment of the first (and vice versa), the assumptions on the environment must be taken into account somehow in this composition.

5.1.3 Other issues

There are other possibilities for future work, that mostly serve the development of tool support for the specification method. This can be quite time consuming, so it might be advisable to fine-tune the method by a number of larger case studies (although prototype

tool support, like the use of PVS might be essential for the feasibility of the case studies). We now list these directions:

1. A user friendly interface, for instance Emacs-forms that are automatically translated into a PVS specification, and visualisation techniques to inspect the PVS specification. Ideally, PVS (or another proof tool) would be completely hidden from the user, so feedback from failed type checking and theorem proving should be translated back to the user interface. Also, the proposed requirements specification method could be integrated with popular engineering approaches, like UML and STATEMATE.

2. Development of proof strategies, in order to solve the proof verification obligations. As the type of theorems is known, one can conceive general strategies for solving them. It might well be the case that other theorem provers have better capabilities for defining strategies. Maybe automated theorem proving, e.g. combinations of resolution and term rewriting, can be applied, provided the specification uses a suitably simple format.

3. The use of common notations for parts of the specification. For instance, the state, invariants and transitions can be specified by Z-schemes, which would immediately provide some form of modularity. States, invariants and a protocol could be directly specified in TLA, etc. It can be tried to cover the whole method with existing methods, so that existing tools can be used. The states/protocol could also be specified in Statecharts.

4. A specification simulator could be built. This would greatly improve the validation abilities of the specification. As we have quite declarative and abstract specifications, it will not be directly executable. However, after some transformations (e.g. making all domains finite and eliminating $\forall$ and $\exists$ quantifiers), simulation in for instance the μCRL-toolkit [GP94] seems possible. In this case, a transition would translate to the process definition (D_k denotes the types of the parameters of input event i_k):

   $$\begin{aligned} M(X:State) = \\ &\Sigma_{\vec{p}:D_1}.\,\Sigma_{Y:State}.\,[i_1(\vec{p})\cdot M(Y)] \triangleleft Pre_1(X) \wedge \Delta_1(X,Y) \wedge Invariant(Y) \triangleright \delta \\ &+ \Sigma_{\vec{p}:D_2}.\,\Sigma_{Y:State}.\,[i_2(\vec{p})\cdot M(Y)] \triangleleft Pre_2(X) \wedge \Delta_2(X,Y) \wedge Invariant(Y) \triangleright \delta \\ &\ldots \\ &+ \Sigma_{\vec{p}:D_n}.\,\Sigma_{Y:State}.\,[i_n(\vec{p})\cdot M(Y)] \triangleleft Pre_n(X) \wedge \Delta_n(X,Y) \wedge Invariant(Y) \triangleright \delta \end{aligned}$$

   This models that $M(X)$ can perform i_k with parameters $\vec{p}$, and then proceed to state $M(Y)$, provided certain conditions hold (between $\triangleleft$ $\triangleright$). The machine would then be modeled as the process $\Sigma_{X:State} M(X) \triangleleft Initial(X) \wedge Invariant(X) \triangleright \delta$. This translation doesn't yet capture the ouput events.

5.2 Evaluation

The goal of the research was to find a method for the specification and analysis of requirements on realistic command and control systems. The tangible result of this project is a template (A.2) with a pseudo PVS specification, which can be filled in for a particular requirements specification. The specification blocks are explained in section 2.2.2. These blocks can be translated onto a machine model, c.f. 2.1. The theorem prover of PVS (2.4.1) can then be used to type check this specification, and to automatically generate the proof obligations. In this way, the specification can be analyzed (2.3).

In order to arrive at the proposed method, a lot of decisions have been taken. These decisions were motivated by

- The engineering phase: requirements specifiation
- The application domain: command and control systems, or more generally large-scale, distributed, reactive systems.

The engineering phase is taken into account by allowing abstract specifications. The used language is very expressive (higher-order logic). Furthermore, integrity constraints and transitions can be defined declaratively, and details can be deferred to refinements.

The application domain is taken into account be decoupling input and output events. The relation between input and output events in command and control systems is via an internal representation, capturing the state of affairs in the environment. The method is developed with real cases in mind. The case study shows that the method is fit for typical applications.

The use of PVS provided us with tool support already in the research phase. PVS supports the method, by automatically generating the verification proof obligations. It is also used in the case study as a proof tool, verifying that these obligations and several other theorems hold with the largest possible certainty. Although not very hard, a number of proofs were rather time consuming. It is expected that especially for the verification proof obligations, dedicated proof commands can be programmed, which make the verification more feasible.

It is not possible to estimate afterwards the time needed for the specification of the case study. The time spent in the project is mixed with the time needed to develop the method. A completely new case study should be carried out, preferably by other people, in order to verify feasibility of the method in terms of human effort.

It is advised that a number of large scale case studies will be conducted by Signaal, or in close collaboration with Signaal, or possibly in other companies, in order to assess the feasibility of the method and to fine tune it. Based on these experiences, and based on current research on the further design process, research should be conducted along the lines of Section 5.1.

Appendix A

Templates

A.1 Formalization of the machine model

The following theory defines the framework for specifications. It can be imported by a concrete specification, by giving actual values to the parameters. It then generates the verification proof obligations. Moreover, it defines the reachability predicate.

```
machine[  I,O,S: TYPE,
          Imap: [I -> pred[[S,S]]],
          Omap: [O -> pred[[S,S]]],
          init: pred[S],
          inv: pred[S],
          pre: [I->pred[S]]
       ]: THEORY

BEGIN

ASSUMING
  i: VAR I
  x,y: VAR S

  init: ASSUMPTION  EXISTS x: init(x) & inv(x)

  no_deadlock: ASSUMPTION
     pre(i)(x) & inv(x) => EXISTS y: inv(y) & Imap(i)(x,y)

ENDASSUMING

Reachable(x):INDUCTIVE bool =
   init(x) & inv(x)
OR EXISTS y,i : Reachable(y) & pre(i)(y) & inv(x) & Imap(i)(y,x)

Invariant_holds: LEMMA  FORALL x: Reachable(x) => inv(x)

END machine
```

A.2 A template for concrete specifications

The template in this section is not a complete PVS specification. It is meant as a starting point for new specifications. Actual details have to filled in on the ...'s. The verification proof obligations are generated by the final `IMPORTING` clause. If this is proved, and all generated type check conditions are proved, then the specification is regarded as correct. The order of the blocks can be changed at will, as long as every identifier is defined before it its first use, which is a PVS requirement.

```
specification [ ... parameters ... ] : THEORY
BEGIN

ASSUMING
  A_1: ASSUMPTION ... formula ...
  ...
  A_j: ASSUMPTION ... formula ...
ENDASSUMING

%%%%%%%%%%%%%%%%%%%%%%%
% A. STATIC INTERFACE %
%%%%%%%%%%%%%%%%%%%%%%%

% 1. Basic Types
%%%%%%%%%%%%%%%%

... type declarations ...

T_1 : TYPE = ... type expression ...
...
T_k : TYPE = ... type expression ...

% 2. Input events
%%%%%%%%%%%%%%%%

IEvents: DATATYPE
BEGIN
  ie_1(...i_params_1...): ie_1?
  ...
  ie_n(...i_params_n...): ie_n?
END IEvents

% 3. Output events
%%%%%%%%%%%%%%%%%

OEvents: DATATYPE
BEGIN
  oe_1(...o_params_1...): oe_1?
  ...
  oe_p(...o_params_p...): oe_p?
END OEvents
```

```
%%%%%%%%%%%%%%%%%%%%%%%%
% B. INFORMATION MODEL %
%%%%%%%%%%%%%%%%%%%%%%%%

% 4. State variables
%%%%%%%%%%%%%%%%%%%%%

State: TYPE =
  [# v_1: T_1,
     ...
     v_k: T_k
  #]

X,Y: VAR State
ie: VAR IEvents
oe: VAR OEvents


% 5. Integrity constraints
%%%%%%%%%%%%%%%%%%%%%%%%%

constraint_1(X):bool = ... boolean expression ...
...
constraint_m(X):bool = ... boolean expression ...

Invariant(X):bool =
  constraint_1(X) & ... & constraint_m(X)

% 6. Initial state
%%%%%%%%%%%%%%%%%%

initial(X):bool = ...


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% C. MAPPING INPUT TO OUTPUT %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% 7. State transitions
%%%%%%%%%%%%%%%%%%%%%%%

Transition_1(i_params_1)(X,Y):bool = ... boolean expression ...
...
Transition_n(i_params_n)(X,Y):bool = ... boolean expression ...

% 8. Output triggers
%%%%%%%%%%%%%%%%%%%%
```

```
Trigger_1(o_params_1)(X,Y):bool = ... boolean expression ...
...
Trigger_p(o_params_p)(X,Y):bool = ... boolean expression ...

% 9. Mapping of inputs to state transitions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Input_table(ie):[State,State->bool] =
  CASES ie OF
    ie_1(i_params_1) : Transition_1(i_params_1),
    ...
    ie_n(i_params_n) : Transition_n(i_params_n)
  ENDCASES

% 10. Mapping of state changes to triggers
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Output_table(oe):[State,State->bool] =
  CASES oe OF
    oe_1(o_params_1) : Trigger_1(o_params_1),
    ...
    oe_p(o_params_p) : Trigger_p(o_params_p)
  ENDCASES


%%%%%%%%%%%%%%%%%%%
% D. ENVIRONMENT %
%%%%%%%%%%%%%%%%%%%

% 11. Preconditions
%%%%%%%%%%%%%%%%%%%%%

Precondition(ie)(X):bool =
  CASES ie OF
    ie_1(i_params_1) : ... boolean expression ...
    ...
    ie_n(i_params_n) : ... boolean expression ...
  ENDCASES


%%%%%%%%%%%%
% ANALYSIS %
%%%%%%%%%%%%

% Check preconditions
%%%%%%%%%%%%%%%%%%%%%%%

ie_1_next: LEMMA
```

```
   Invariant(X) & Precondition(ie_1(i_params_1))(X)
=> EXISTS Y: Invariant(Y) & Transition_1(i_params_1)(X,Y)

...

ie_n_next: LEMMA
   Invariant(X) & Precondition(ie_n(i_params_n))(X)
=> EXISTS Y: Invariant(Y) & Transition_n(i_params_n)(X,Y)

% Generate verification proof obligations
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

IMPORTING machine[IEvents,OEvents,State,
                  Input_table,Output_table,
                  initial,Invariant,Precondition]


% Putative Theorems
%%%%%%%%%%%%%%%%%%%

...

END specification
```

Appendix B

Complete PVS specifications

B.1 Top specification

```
spec1: THEORY
BEGIN

%%%%%%%%%%%%%%%%%%%%%%%%
% A. STATIC INTERFACE %
%%%%%%%%%%%%%%%%%%%%%%%%

% 1. Basic Types
%%%%%%%%%%%%%%%%

Sensor_track, System_track: TYPE
Sensor_track_state, System_track_state: TYPE+

% 2. Input events
%%%%%%%%%%%%%%%%%

IEvents: DATATYPE
BEGIN
  new(sn:Sensor_track,s:Sensor_track_state): new_sens?
  update(sn:Sensor_track,s:Sensor_track_state): update_sens?
  wipe(sn:Sensor_track): wipe_sens?
END IEvents

% 3. Output events
%%%%%%%%%%%%%%%%%%

OEvents: DATATYPE
BEGIN
  new(tn:System_track,t:System_track_state): new_sys?
  update(tn:System_track,t:System_track_state): update_sys?
  wipe(tn:System_track): wipe_sys?
  warn(sn:Sensor_track): warn_sys?
```

```
END OEvents


%%%%%%%%%%%%%%%%%%%%%%%%
% B. INFORMATION MODEL %
%%%%%%%%%%%%%%%%%%%%%%%%

% 4. State variables
%%%%%%%%%%%%%%%%%%%%

State: TYPE =
  [# sensor_ids: setof[Sensor_track],
     system_ids: setof[System_track],
     system_info: [System_track->System_track_state],
     joined: pred[[Sensor_track,System_track]]
  #]

  s,s1: VAR Sensor_track_state
  sn,sn1: VAR Sensor_track
  t: VAR System_track_state
  tn,tn1,tn2: VAR System_track
  ie: VAR IEvents
  oe: VAR OEvents
  X,Y: VAR State

% 5. Integrity constraints
%%%%%%%%%%%%%%%%%%%%%%%%%%

constraint1(X):bool = FORALL sn,tn :
   joined(X)(sn,tn) => sensor_ids(X)(sn) & system_ids(X)(tn)
constraint2(X):bool = FORALL sn:
  sensor_ids(X)(sn) => exists1! tn: joined(X)(sn,tn)
constraint3(X):bool = FORALL tn:
  system_ids(X)(tn) => EXISTS sn: joined(X)(sn,tn)

Invariant(X):bool =
  constraint1(X) & constraint2(X) & constraint3(X)

% 6. Initial state
%%%%%%%%%%%%%%%%%%

initial(X):bool = empty?(sensor_ids(X))


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% C. MAPPING INPUT TO OUTPUT %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% 7. State transitions
```

```
%%%%%%%%%%%%%%%%%%%%%

correlates(s,t):bool
decorrelates(s,t):bool

corex   : AXIOM  EXISTS t : correlates(s,t)
cordecor: AXIOM  NOT (correlates(s,t) & decorrelates(s,t))

new_sensor_track(sn,s)(X,Y):bool =
  sensor_ids(Y) = add(sn,sensor_ids(X))
& FORALL tn: joined(Y)(sn,tn)
             => correlates(s,system_info(Y)(tn))

wipe_sensor_track(sn)(X,Y):bool =
  sensor_ids(Y) = remove(sn,sensor_ids(X))
& subset?(system_ids(Y),system_ids(X))
& subset?(joined(Y),joined(X))

update_sensor_track(sn,s)(X,Y):bool =
  sensor_ids(Y) = sensor_ids(X)
& FORALL tn: decorrelates(s,system_info(Y)(tn))
             => NOT joined(Y)(sn,tn)

% 8. Output triggers
%%%%%%%%%%%%%%%%%%%%

new_system_trigger(tn,t)(X,Y):bool
=   system_ids(Y)(tn)
  & NOT system_ids(X)(tn)
  & t=system_info(Y)(tn)

update_system_trigger(tn,t)(X,Y):bool
=   system_ids(X)(tn)
  & system_ids(Y)(tn)
  & system_info(X)(tn) /= system_info(Y)(tn)
  & t = system_info(Y)(tn)

wipe_system_trigger(tn)(X,Y):bool
=   system_ids(X)(tn)
  & NOT system_ids(Y)(tn)

detect_decorrelation(sn)(X,Y):bool =
 EXISTS tn1,tn2 :
    joined(X)(sn,tn1)
  & joined(Y)(sn,tn2)
  & tn1 /= tn2

% 9. Mapping of inputs to state transitions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

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

% 10. Mapping of state changes to triggers
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Output_table(oe):[State,State->bool] =
  CASES oe OF
    new(tn,t)   : new_system_trigger(tn,t),
    update(tn,t): update_system_trigger(tn,t),
    wipe(tn)    : wipe_system_trigger(tn),
    warn(sn)    : detect_decorrelation(sn)
  ENDCASES


%%%%%%%%%%%%%%%%%%
% D. ENVIRONMENT %
%%%%%%%%%%%%%%%%%%

% 11. Preconditions
%%%%%%%%%%%%%%%%%%%

Precondition(ie)(X):bool =
  CASES ie OF
    new(sn,s)    :   (NOT sensor_ids(X)(sn))
                   & (EXISTS tn: NOT system_ids(X)(tn)),
    update(sn,s) :   sensor_ids(X)(sn)
                   & (EXISTS tn: NOT system_ids(X)(tn)),
    wipe(sn)     : sensor_ids(X)(sn)
  ENDCASES


%%%%%%%%%%%%
% ANALYSIS %
%%%%%%%%%%%%

% Check preconditions
%%%%%%%%%%%%%%%%%%%%%

new_next: LEMMA
   Invariant(X) & Precondition(new(sn,s))(X)
=> EXISTS Y: Invariant(Y) & new_sensor_track(sn,s)(X,Y)

wipe_next: LEMMA
```

```
  Invariant(X) & Precondition(wipe(sn))(X)
=> EXISTS Y: Invariant(Y) & wipe_sensor_track(sn)(X,Y)

update_next: LEMMA
  Invariant(X) & Precondition(update(sn,s))(X)
=> EXISTS Y: Invariant(Y) & update_sensor_track(sn,s)(X,Y)

% Generate verification proof obligations
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

IMPORTING machine[IEvents,OEvents,State,
                  Input_table,Output_table,
                  initial,Invariant,Precondition]

END spec1
```

B.2 First refinement

```
spec2
 [ Sensor_track, Sensor_track_state,
   System_track, System_track_state: TYPE+,
   correlates, decorrelates:[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

ASSUMING

s: VAR Sensor_track_state
t: VAR System_track_state

cordecor: ASSUMPTION  NOT (correlates(s,t) & decorrelates(s,t))
corin:    ASSUMPTION  correlates(s,initiate(s))
decorup:  ASSUMPTION  decorrelates(s,update(s,t)) => decorrelates(s,t)

ENDASSUMING


%%%%%%%%%%%%%%%%%%%%%%%%
% A. STATIC INTERFACE %
%%%%%%%%%%%%%%%%%%%%%%%%

% 1. Basic Types
%%%%%%%%%%%%%%%%%

% See the parameters of the theory

% 2. Input events
```

```
%%%%%%%%%%%%%%%%%

IEvents: DATATYPE
BEGIN
  new(sn:Sensor_track,s:Sensor_track_state): new_sens?
  update(sn:Sensor_track,s:Sensor_track_state): update_sens?
  wipe(sn:Sensor_track): wipe_sens?
END IEvents

% 3. Output events
%%%%%%%%%%%%%%%%%%

OEvents: DATATYPE
BEGIN
  new(tn:System_track,t:System_track_state): new_sys?
  update(tn:System_track,t:System_track_state): update_sys?
  wipe(tn:System_track): wipe_sys?
  warn(sn:Sensor_track): warn_sys?
END OEvents


%%%%%%%%%%%%%%%%%%%%%%%%
% B. INFORMATION MODEL %
%%%%%%%%%%%%%%%%%%%%%%%%

% 4. State variables
%%%%%%%%%%%%%%%%%%%%

State: TYPE =
  [# sensor_ids: setof[Sensor_track],
     system_ids: setof[System_track],
     system_info: [(system_ids)->System_track_state],
     joined: [(sensor_ids)->(system_ids)]
  #]

tn: VAR System_track
sn: VAR Sensor_track
X,Y: VAR State
ie: VAR IEvents
oe: VAR OEvents

% 5. Integrity constraints
%%%%%%%%%%%%%%%%%%%%%%%%%%

% No constraints

Invariant(X):bool = TRUE

% 6. Initial state
```

```
%%%%%%%%%%%%%%%%%

initial(X):bool = empty?(sensor_ids(X)) & empty?(system_ids(X))


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% C. MAPPING INPUT TO OUTPUT %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% 7. State transitions
%%%%%%%%%%%%%%%%%%%%%%

% The main transitions are:
% - new_sensor_track
% - update_sensor_track
% - wipe_sensor_track

no_correlation(s,X):bool =
  FORALL tn: system_ids(X)(tn)
             => NOT correlates(s,system_info(X)(tn))

correlate_to_system_track(sn,s)(X,Y):bool =
  EXISTS tn:
    system_ids(X)(tn)
  & correlates(s,system_info(X)(tn))
  & Y = X WITH
    [sensor_ids := add(sn,sensor_ids(X)),
     joined := joined(X) WITH [sn:=tn],
     system_info := system_info(X)
        WITH [tn:=update(s,system_info(X)(tn))]]

new_system_track(sn,s)(X,Y):bool =
  no_correlation(s,X)
& EXISTS tn:
    (NOT system_ids(X)(tn))
  & Y =
    (# sensor_ids := add(sn,sensor_ids(X)),
       system_ids := add(tn,system_ids(X)),
       joined := joined(X) WITH [sn:=tn],
       system_info := system_info(X) WITH [tn := initiate(s)] #)

new_sensor_track(sn,s)(X,Y):bool =
   new_system_track(sn,s)(X,Y)
OR correlate_to_system_track(sn,s)(X,Y)

filter(X):State =
  X WITH
    [system_ids := {tn | system_ids(X)(tn)
       & EXISTS sn: sensor_ids(X)(sn) & tn=joined(X)(sn)},
```

```
     system_info := restrict(system_info(X))]

wipe_sensor_track(sn)(X,Y):bool =
  sensor_ids(X)(sn)
& Y = filter(X WITH [
   sensor_ids := remove(sn,sensor_ids(X)),
   joined := restrict(joined(X))])

update_sensor_track(sn,s)(X,Y):bool =
  sensor_ids(X)(sn)
& LET tn = joined(X)(sn), t = system_info(X)(tn) IN
  IF decorrelates(s,t)
  THEN EXISTS (Z:State): new_sensor_track(sn,s)(X,Z) & Y=filter(Z)
  ELSE Y = X WITH
           [system_info:= system_info(X) WITH [tn := update(s,t)]]
  ENDIF

% 8. Output triggers
%%%%%%%%%%%%%%%%%%%

new_system_trigger(tn,t)(X,Y):bool =
    system_ids(Y)(tn)
& NOT system_ids(X)(tn)
& t=system_info(Y)(tn)

update_system_trigger(tn,t)(X,Y):bool =
  system_ids(X)(tn)
& system_ids(Y)(tn)
& system_info(X)(tn) /= system_info(Y)(tn)
& t = system_info(Y)(tn)

wipe_system_trigger(tn)(X,Y):bool =
  system_ids(X)(tn)
& NOT system_ids(Y)(tn)

detect_decorrelation(sn)(X,Y):bool =
  sensor_ids(X)(sn) & sensor_ids(Y)(sn)
& joined(X)(sn) /= joined(Y)(sn)

% 9. Mapping of inputs to state transitions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

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

```
% 10. Mapping of state changes to triggers
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Output_table(oe):[State,State->bool] =
    CASES oe OF
    new(tn,t)   : new_system_trigger(tn,t),
    update(tn,t): update_system_trigger(tn,t),
    wipe(tn)    : wipe_system_trigger(tn),
    warn(sn)    : detect_decorrelation(sn)
  ENDCASES


%%%%%%%%%%%%%%%%%%
% D. ENVIRONMENT %
%%%%%%%%%%%%%%%%%%

% 11. Preconditions
%%%%%%%%%%%%%%%%%%%%

Precondition(ie)(X):bool =
  CASES ie OF
    new(sn,s)    :   (NOT sensor_ids(X)(sn))
                   & (EXISTS tn: NOT system_ids(X)(tn)),
    update(sn,s) :   sensor_ids(X)(sn)
                   & (EXISTS tn: NOT system_ids(X)(tn)),
    wipe(sn)     : sensor_ids(X)(sn)
  ENDCASES


%%%%%%%%%%%
% ANALYSIS %
%%%%%%%%%%%


% Check: Checking preconditions of function model
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

e1: LEMMA system_ids(X)(tn) & correlates(s,system_info(X)(tn)) =>
     EXISTS Y: correlate_to_system_track(sn,s)(X,Y)

e2: LEMMA no_correlation(s,X) & (EXISTS tn: NOT system_ids(X)(tn))
    => EXISTS Y: new_system_track(sn,s)(X,Y)

e3: LEMMA (EXISTS tn: NOT system_ids(X)(tn))=>
           EXISTS Y: new_sensor_track(sn,s)(X,Y)

e4: LEMMA sensor_ids(X)(sn) &  (EXISTS tn: NOT system_ids(X)(tn))
    => EXISTS Y: update_sensor_track(sn,s)(X,Y)
```

```
e5: LEMMA sensor_ids(X)(sn)
    => EXISTS Y: wipe_sensor_track(sn)(X,Y)

% Generate verification proof obligations
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

IMPORTING machine[IEvents,OEvents,State,
                  Input_table,Output_table,
                  initial,Invariant,Precondition]


% Check: constraint1 and constraint2 hold
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

constraint1: LEMMA
   sensor_ids(X)(sn) & tn = joined(X)(sn) => system_ids(X)(tn)

constraint2: LEMMA
  sensor_ids(X)(sn) => exists1! tn: tn = joined(X)(sn)


% Check: constraint3 is an invariant
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

constraint3(X):bool = FORALL tn:
  system_ids(X)(tn) =>
    EXISTS sn: sensor_ids(X)(sn) & tn = joined(X)(sn)

l1: LEMMA constraint3(X) & new_system_track(sn,s)(X,Y)
          & (NOT sensor_ids(X)(sn)) => constraint3(Y)

l2: LEMMA constraint3(X) & correlate_to_system_track(sn,s)(X,Y)
          & (NOT sensor_ids(X)(sn)) => constraint3(Y)

l3: LEMMA constraint3(filter(X))

l4: LEMMA constraint3(X) & new_sensor_track(sn,s)(X,Y)
          & (NOT sensor_ids(X)(sn)) => constraint3(Y)

l5: LEMMA constraint3(X) & update_sensor_track(sn,s)(X,Y)
          => constraint3(Y)

l6: LEMMA constraint3(X) & wipe_sensor_track(sn)(X,Y)
          => constraint3(Y)

constraint3: LEMMA Reachable(X) => constraint3(X)


% Check: updating doesn't lead to decorrelation
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

update_no_decorrelation: LEMMA
  update_sensor_track(sn,s)(X,Y) =>
    NOT decorrelates(s,system_info(Y)(joined(Y)(sn)))


% Check: simultaneous input and output events
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

oe1,oe2: VAR OEvents
OE: VAR setof[OEvents]
tn1,tn2: VAR System_track
t1,t2: VAR System_track_state
sn1: VAR Sensor_track

May_cause(ie,OE):bool = EXISTS X,Y:
    Precondition(ie)(X)
  & Input_table(ie)(X,Y)
  & forall oe: OE(oe) => Output_table(oe)(X,Y)

May_appear_together(OE):bool =
  EXISTS ie: May_cause(ie,OE)

CONVERSION singleton[OEvents]

L1: LEMMA NOT May_cause(new(sn,s), wipe(tn))
L2: LEMMA NOT May_cause(wipe(sn), new(tn,t))
L3: LEMMA NOT May_cause(wipe(sn), update(tn,t))
L4: LEMMA NOT May_cause(wipe(sn), warn(sn1))
L5: LEMMA NOT May_appear_together( (: new(tn1,t1),update(tn2,t2) :))

END spec2
```

B.3 Second refinement

```
spec3
 [ Sensor_track, System_track,
   Source: TYPE+,
   Margin1,Margin2: posreal
 ]: THEORY
BEGIN

ASSUMING
  Margins: ASSUMPTION Margin1 < Margin2
ENDASSUMING

Identification: TYPE = {friend, hostile, pending}
Kinetic: TYPE = [# px,py,pz: real #]
```

```
Sensor_track_state: TYPE =
[# source: Source,
   identification: Identification,
   kinetics: Kinetic
#]

System_track_state: TYPE =
[# identification: Identification,
   kinetics: Kinetic
#]

s: VAR Sensor_track_state
t: VAR System_track_state

initiate(s):System_track_state =
  (# identification := identification(s), kinetics:=kinetics(s) #)

mean(s,t): Kinetic =
  LET k1 = kinetics(s), k2 = kinetics(t) IN
  (# px := (px(k1) + px(k2)) / 2,
     py := (py(k1) + py(k2)) / 2,
     pz := (pz(k1) + pz(k2)) / 2
   #)

update(s,t):System_track_state =
  (# identification := identification(s), kinetics:=mean(s,t) #)

distance(s,t):real =
  LET k1 = kinetics(s), k2 = kinetics(t) IN
    (px(k1)-px(k2)) * (px(k1)-px(k2))
   +(py(k1)-py(k2)) * (py(k1)-py(k2))
   +(pz(k1)-pz(k2)) * (pz(k1)-pz(k2))

conflicting(s,t):bool =
  LET id1 = identification(s), id2 = identification(t) IN
TABLE
  id1, id2 |[pending| friend |hostile]|
%-----------------------------------
|pending   | FALSE  |  FALSE | FALSE ||
|friend    | FALSE  |  FALSE | TRUE  ||
|hostile   | FALSE  |  TRUE  | FALSE ||
ENDTABLE%-----------------------------

correlates(s,t):bool =
    distance(s,t) <= Margin1 * Margin1
  & NOT conflicting(s,t)

decorrelates(s,t):bool =
```

```
     distance(s,t) >= Margin2 * Margin2
  OR conflicting(s,t)

IMPORTING spec2
  [Sensor_track,Sensor_track_state,
   System_track,System_track_state,
   correlates,decorrelates,initiate,update]

END spec3
```

Appendix C

Proof status reports

```
 Proof summary for theory machine
    Invariant_holds.............................proved - complete
    Theory totals: 1 formulas, 1 attempted, 1 succeeded.

Grand Totals: 1 proofs, 1 attempted, 1 succeeded.
```

```
 Proof summary for theory spec1
    new_next....................................proved - complete
    wipe_next...................................proved - complete
    update_next.................................proved - complete
    IMPORTING1_TCC1.............................proved - complete
    IMPORTING1_TCC2.............................proved - complete
    Theory totals: 5 formulas, 5 attempted, 5 succeeded.

Grand Totals: 5 proofs, 5 attempted, 5 succeeded.
```

```
 Proof summary for theory spec2
    correlate_to_system_track_TCC1.....................proved - complete
    new_system_track_TCC1..............................proved - complete
    new_system_track_TCC2..............................proved - complete
    filter_TCC1........................................proved - complete
    filter_TCC2........................................proved - complete
    wipe_sensor_track_TCC1.............................proved - complete
    update_sensor_track_TCC1...........................proved - complete
    e1.................................................proved - complete
    e2.................................................proved - complete
    e3.................................................proved - complete
    e4.................................................proved - complete
    e5.................................................proved - complete
    IMPORTING1_TCC1....................................proved - complete
    IMPORTING1_TCC2....................................proved - complete
    constraint1........................................proved - complete
    constraint2........................................proved - complete
    l1.................................................proved - complete
    l2.................................................proved - complete
    l3.................................................proved - complete
    l4.................................................proved - complete
    l5.................................................proved - complete
    l6.................................................proved - complete
    constraint3........................................proved - complete
    update_no_decorrelation_TCC1.......................proved - complete
    update_no_decorrelation............................proved - complete
    L1.................................................proved - complete
    L2.................................................proved - complete
    L3.................................................proved - complete
    L4.................................................proved - complete
    L5.................................................proved - complete
    Theory totals: 30 formulas, 30 attempted, 30 succeeded.

Grand Totals: 30 proofs, 30 attempted, 30 succeeded.
```

```
 Proof summary for theory spec3
    IMPORTING1_TCC1....................................proved - complete
    IMPORTING1_TCC2....................................proved - complete
    IMPORTING1_TCC3....................................proved - complete
    Theory totals: 3 formulas, 3 attempted, 3 succeeded.

Grand Totals: 3 proofs, 3 attempted, 3 succeeded.
```

Appendix D

Complete PVS proofs

D.1 General machine

```
Proof scripts for theory machine:

machine.Invariant_holds: proved - complete
("" (USE "init") (EXPAND "Reachable") (GRIND))
```

D.2 Top specification

```
Proof scripts for theory spec1:

spec1.new_next: proved - complete
(""
 (SKOSIMP*)
 (USE "corex")
 (EXPAND "Precondition")
 (SKOSIMP*)
 (INST 3 "(# sensor_ids := add(sn!1,sensor_ids(X!1)),
             system_ids := add(tn!1,system_ids(X!1)),
             joined := add((sn!1,tn!1),joined(X!1)),
             system_info := system_info(X!1) WITH [tn!1:= t!1] #)")
 (SPLIT)
 (("1"
   (EXPAND "Invariant")
   (FLATTEN)
   (SPLIT)
   (("1" (EXPAND "constraint1") (SKOSIMP*) (GRIND))
    ("2"
     (GRIND :IF-MATCH NIL)
     (("1" (GRIND)) ("2" (GRIND)) ("3" (GRIND)) ("4" (GRIND)) ("5" (GRIND))
      ("6" (INST? -4) (GRIND))))
    ("3" (GRIND :IF-MATCH NIL) (("1" (GRIND)) ("2" (GRIND))))))
  ("2" (GRIND))))
```

```
spec1.wipe_next: proved - complete
(""
 (SKOSIMP*)
 (INST 1 "LET del = {tn | system_ids(X!1)(tn)
                   & FORALL sn: joined(X!1)(sn,tn) => sn=sn!1} IN
 (# sensor_ids := remove(sn!1,sensor_ids(X!1)),
    system_ids := {tn | system_ids(X!1)(tn) & NOT del(tn)},
    joined := {(sn,tn) |
                joined(X!1)(sn,tn) & sn/=sn!1 & NOT del(tn)},
    system_info := system_info(X!1) #)")
 (SPLIT)
 (("1"
   (GRIND :IF-MATCH NIL)
   (("1" (GRIND)) ("2" (GRIND)) ("3" (GRIND)) ("4" (GRIND)) ("5" (GRIND))
    ("6" (GRIND)) ("7" (GRIND)) ("8" (GRIND))))
  ("2" (GRIND))))

spec1.update_next: proved - complete
(""
 (SKOSIMP*)
 (USE "wipe_next")
 (GROUND)
 (("1"
   (SKOSIMP*)
   (USE "new_next" ("X" "Y!1"))
   (GROUND)
   (("1"
     (SKOSIMP*)
     (INST 1 "Y!2")
     (ASSERT)
     (EXPAND "Precondition")
     (FLATTEN)
     (HIDE -1 -3 -5 -7)
     (LEMMA "cordecor")
     (GRIND)
     (APPLY-EXTENSIONALITY :HIDE? T)
     (GRIND))
    ("2" (GRIND))))
  ("2" (EXPAND "Precondition") (FLATTEN) (PROPAX))))

spec1.IMPORTING1_TCC1: proved - complete
(""
 (INST 1 "(# sensor_ids  := emptyset,
     system_ids  := emptyset,
     joined      := emptyset,
     system_info := LAMBDA tn : epsilon! t : TRUE
     #)")
 (GRIND))
```

```
spec1.IMPORTING1_TCC2: proved - complete
(""
 (INDUCT "i")
 (("1" (USE "new_next") (GRIND)) ("2" (USE "update_next") (GRIND))
  ("3" (USE "wipe_next") (GRIND))))
```

D.3 First refinement

```
Proof scripts for theory spec2:

spec2.correlate_to_system_track_TCC1: proved - complete
("" (SUBTYPE-TCC))

spec2.new_system_track_TCC1: proved - complete
("" (SUBTYPE-TCC))

spec2.new_system_track_TCC2: proved - complete
("" (SUBTYPE-TCC))

spec2.filter_TCC1: proved - complete
("" (SUBTYPE-TCC))

spec2.filter_TCC2: proved - complete
("" (SUBTYPE-TCC))

spec2.wipe_sensor_track_TCC1: proved - complete
("" (SUBTYPE-TCC))

spec2.update_sensor_track_TCC1: proved - complete
("" (SUBTYPE-TCC))

spec2.e1: proved - complete
(""
 (SKOSIMP*)
 (INST 1 "X!1 WITH
 [sensor_ids := add(sn!1,sensor_ids(X!1)),
  joined := joined(X!1) WITH [sn!1:=tn!1],
  system_info := system_info(X!1)
    WITH [tn!1:=update(s!1,system_info(X!1)(tn!1))] ]")
 (("1" (GRIND)) ("2" (GRIND))))

spec2.e2: proved - complete
(""
 (SKOSIMP*)
 (INST 2 "(#
 sensor_ids := add(sn!1,sensor_ids(X!1)),
 system_ids := add(tn!1,system_ids(X!1)),
 joined := joined(X!1) WITH [sn!1:=tn!1],
 system_info := system_info(X!1) WITH [tn!1 := initiate(s!1)] #)")
```

```
 (("1" (GRIND)) ("2" (GRIND)) ("3" (GRIND))))

spec2.e3: proved - complete
(""
 (SKOSIMP*)
 (CASE "EXISTS tn: system_ids(X!1)(tn)
                     & correlates(s!1,system_info(X!1)(tn))")
 (("1" (HIDE 1) (SKOSIMP*) (USE "e1") (PROP) (SKOSIMP) (INST 1 "Y!1") (GRIND))
  ("2"
   (USE "e2")
   (SPLIT -1)
   (("1" (SKOSIMP) (INST 3 "Y!1") (EXPAND "new_sensor_track") (GROUND))
    ("2" (HIDE 3 4) (GRIND)) ("3" (GRIND))))))

spec2.e4: proved - complete
(""
 (SKOSIMP*)
 (CASE "decorrelates(s!1,system_info(X!1)(joined(X!1)(sn!1)))")
 (("1"
   (USE "e3")
   (SPLIT -1)
   (("1"
     (SKOSIMP)
     (INST 2 "filter(Y!1)")
     (EXPAND "update_sensor_track")
     (ASSERT)
     (INST 2 "Y!1")
     (ASSERT))
    ("2" (INST 1 "tn!1"))))
  ("2"
   (INST 3 "X!1 WITH
  [system_info:=system_info(X!1) WITH
    [(joined(X!1)(sn!1)):=update(s!1,system_info(X!1)(joined(X!1)(sn!1)))]]")
   (GRIND))
  ("3" (PROPAX))))

spec2.e5: proved - complete
(""
 (SKOSIMP*)
 (INST 1 "filter(X!1 WITH [ sensor_ids := remove(sn!1,sensor_ids(X!1)),
   joined := restrict(joined(X!1))])")
 (("1" (GRIND)) ("2" (GRIND))))

spec2.IMPORTING1_TCC1: proved - complete
(""
 (INST 1 "(# sensor_ids  := emptyset,
     system_ids  := emptyset,
     joined      := LAMBDA (sn:(emptyset[Sensor_track])):
                      epsilon! (tn:(emptyset[System_track])) : TRUE,
```

```
    system_info := LAMBDA (tn:(emptyset[System_track])):
                      epsilon! (tn:(emptyset[System_track_state])) : TRUE
  #)")
 (("1" (GRIND)) ("2" (GRIND)) ("3" (GRIND))))

spec2.IMPORTING1_TCC2: proved - complete
(""
 (INDUCT "i")
 (("1" (USE "e3") (GRIND)) ("2" (USE "e4") (GRIND)) ("3" (USE "e5") (GRIND))))

spec2.constraint1: proved - complete
("" (GRIND))

spec2.constraint2: proved - complete
("" (GRIND))

spec2.l1: proved - complete
(""
 (EXPAND "new_system_track")
 (SKOSIMP*)
 (REPLACE -3 3 :HIDE? T)
 (EXPAND "constraint3")
 (SKOSIMP*)
 (EXPAND "add")
 (SPLIT)
 (("1" (GRIND)) ("2" (GRIND))))

spec2.l2: proved - complete
(""
 (SKOSIMP*)
 (EXPAND "correlate_to_system_track")
 (SKOSIMP*)
 (EXPAND "constraint3")
 (SKOSIMP*)
 (INST -1 "tn!2")
 (PROP)
 (("1" (SKOSIMP*) (INST? 2) (GRIND)) ("2" (GRIND))))

spec2.l3: proved - complete
("" (GRIND))

spec2.l4: proved - complete
("" (SKOSIMP*) (EXPAND "new_sensor_track") (USE "l1") (USE "l2") (PROP))

spec2.l5: proved - complete
(""
 (SKOSIMP*)
 (EXPAND "update_sensor_track")
 (FLATTEN)
```

```
 (SPLIT -3)
 (("1" (SKOSIMP*) (USE "l3") (GROUND))
  ("2" (FLATTEN) (REPLACE -1 2 :HIDE? T) (EXPAND "constraint3") (PROPAX))))

spec2.l6: proved - complete
(""
 (SKOSIMP*)
 (EXPAND "wipe_sensor_track")
 (USE "l3" :IF-MATCH ALL)
 (("1" (GROUND)) ("2" (GRIND)) ("3" (GRIND))))

spec2.constraint3: proved - complete
(""
 (RULE-INDUCT "Reachable")
 (SKOSIMP*)
 (SPLIT)
 (("1" (GRIND))
  ("2"
   (CASE "NOT FORALL (y: State), (i: IEvents):
        NOT (constraint3(y) & Precondition(i)(y) & Input_table(i)(y,x!1))")
   (("1"
     (HIDE -1)
     (INDUCT "i")
     (("1"
       (SKOSIMP*)
       (LEMMA "l4")
       (INST -1 "y!1" "x!1" "new2_var!1" "new1_var!1")
       (GRIND))
      ("2"
       (LEMMA "l5")
       (SKOSIMP*)
       (INST -1 "y!1" "x!1" "update2_var!1" "update1_var!1")
       (GRIND))
      ("3"
       (SKOSIMP*)
       (LEMMA "l6")
       (INST -1 "y!1" "x!1" "wipe1_var!1")
       (GRIND))))
    ("2" (SKOSIMP*) (INST?) (PROP))))))

spec2.update_no_decorrelation_TCC1: proved - complete
("" (SUBTYPE-TCC))

spec2.update_no_decorrelation: proved - complete
(""
 (GRIND :IF-MATCH NIL)
 (("1" (USE "corin") (USE "cordecor") (PROP))
  ("2" (USE "cordecor") (USE "decorup") (PROP)) ("3" (USE "decorup") (PROP))))
```

```
spec2.L1: proved - complete
("" (GRIND))

spec2.L2: proved - complete
("" (GRIND))

spec2.L3: proved - complete
("" (GRIND))

spec2.L4: proved - complete
("" (GRIND))

spec2.L5: proved - complete
(""
 (EXPAND "May_appear_together")
 (EXPAND "May_cause")
 (SKOSIMP*)
 (INST-CP -3 "new(tn1!1,t1!1)")
 (INST -3 "update(tn2!1,t2!1)")
 (GRIND))
```

D.4 Second refinement

```
Proof scripts for theory spec3:

spec3.IMPORTING1_TCC1: proved - complete
("" (USE "Margins") (USE "lt_times_lt_pos1" ("y" "Margin2")) (GRIND))

spec3.IMPORTING1_TCC2: proved - complete
("" (GRIND))

spec3.IMPORTING1_TCC3: proved - complete
("" (GRIND))
```

Appendix E

Further reading

We give a brief overview, where the interested reader can find related literature.

- On similar applications: [Hal96, DS97, MPN+95]
- On requirements engineering: [LK95, Wie96, SS97, Zav97]
- On formal methods: [HB95, Rus93, Rus95, CW96]
- On PVS: [SSJ+96, ORSH95]
- Other formalisms: [Spi92, SBC92, Jon90, BBP87, MP92, Lam94, GP94, BB87, Lyn88, OSRSC98]
- ORKEST publications: [PHJ98, PHJ99]

Bibliography

[BB87] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[BBP87] B. Banieqbal, H. Barringer, and A. Pnueli, editors. *Temporal Logic in Specification*, volume 398 of *LNCS*. Springer, 1987.

[CW96] E.M. Clarke and J.M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[DS97] B. Dutertre and V. Stavridou. Formal requirements analysis of an avionics control system. *IEEE Trans. on SE*, 23(5):267–278, 1997.

[GP94] 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.

[Hal96] A. Hall. Using formal methods to develop an ATC information system. *IEEE Software*, 13(2):66–76, 1996.

[HB95] M.G. Hinchey and J.P. Bowen, editors. *Applications of Formal Methods*. Prentice Hall, 1995.

[HL96] M.P.E. Heimdahl and N.G. Leveson. Completeness and consistency in hierarchical state-based requirements. *IEEE Trans. on SE*, 22(6):363–377, 1996.

[HU80] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, N. Reading, MA, 1980.

[Jon90] C.B. Jones. *Systematic Software Development using VDM*. Prentice Hall, 2nd edition, 1990.

[Lam94] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[LK95] P. Loucopoulos and V. Karakostas. *Systems Requirements Engineering*. McGraw-Hill Book Company Europe, London, 1995.

[Lyn88] 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.

[Mea55] G.H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems – Specification*. Springer, 1992.

[MPN+95] T.L. McCluskey, J.M. Porteous, Y. Naik, C.N. Taylor, and S. Jones. A requirements capture method and its use in an air traffic control application. *Software – Practice and Experience*, 25(1):47–71, 1995.

[ORSH95] S. Owre, J.M. Rushby, N. Shankar, and F. Von Henke. Formal Verification of Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Trans. on SE*, 21(2):107–125, 1995.

[OSRSC98] 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.

[PHJ98] 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.

[PHJ99] J.C. van de Pol, J.J.M. Hooman, and E. de Jong. Modular formal specification of data and behaviour. In *To appear in proc. of IFM'99 (York)*, 1999.

[Ram94] N. Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, 1994.

[Rus93] J. Rushby. Formal methods and the certification of critical systems. Technical Report SRI-CSL-93-7, SRI International, Menlo Park, CA, 1993.

[Rus95] J.M. Rushby. Formal methods and their role in the certification of critical systems. Technical Report CSL-95-01, CSL, 1995.

[SBC92] S. Stepney, R. Barden, and D. Cooper, editors. *Object Orientation in Z*. Workshops in Computing. Springer, 1992.

[Spi92] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 2nd edition, 1992.

[SS97] I. Sommerville and P. Sawyer. *Requirements Engineering*. Wiley, Chichester, 1997.

[SSJ+96] 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.

[Wie96] R.J. Wieringa. *Requirements Engineering: Frameworks for Understanding*. John Wiley, Chichester, 1996.

[Zav97] P. Zave. Classification of research efforts in requirements engineering. *ACM Computing Surveys*, 29(4):315–321, 1997.