# Formal Requirements Specification for Command and Control Systems

Jaco van de Pol(1) Jozef Hooman(1) Edwin de Jong(2)

(1) Eindhoven University of Technology
Dept. of Computing Science
P.O. Box 513, 5600 MB Eindhoven
The Netherlands
Email: {jaco,hooman}@win.tue.nl

(2) Hollandse Signaalapparaten BV
Dept. of Applied System Research
P.O. Box 42, 7550 GD Hengelo
The Netherlands
Email: edejong@signaal.nl

**Abstract**

*This paper presents an approach to formal requirements specification of embedded systems. The specific demands of a specification for command and control systems are addressed. The proposed method allows various views of a system, like conventional methods. The added value lies in the fact that the relationship between the views is specified formally, and consistency between views can be analyzed formally. As a case study, we develop and analyze a formal requirements specification for a subsystem of a realistic command and control system. Specification and verification are carried out using the language and proof checker of PVS.*

## 1 Introduction

**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.

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 then planned, by assigning a time-frame and sufficient resources, and 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.

**Needs.** 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. So it is preferable to detect errors at an early stage of the development process, viz. in the requirements specification phase.

Errors in the specification can also be detected by analysis, but this requires that 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 a *formal* specification helps to detect errors early.

Analysis of formal specifications, however, is very time consuming. Formal methods are cost effective for systems of industrial size under certain conditions only. Firstly, tool support is needed, to make the analysis less labour intensive. Secondly, formal specifications must be modular, modifiable and extensible, in order to allow for an iterative development of the specification. Finally, as argued by [7], formal methods can only be cost effective if the resulting products are reused.

On the other hand, [7] argues that formal specification is essential for the success of reusing software components, so this is another motivation for research on formal specification.

**Goal.** 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.

**Our approach.** The good properties of informal specifications should be maintained, especially their readability and the possibility to have different views of the same system, like in e.g. OMT [10].

In order to maintain readability, we propose to interleave the formal and informal specification by means of a "literate specification" style. To get a well-structured specification, we follow the conventional approach of presenting different views of a system, viz. the information-, function- and control-model. Having different views is an advantage especially if it is clear how the various models interrelate. It is indicated how consistency between these models can be formally stated and verified. This is possible, because we express the various models in the same formal language.

To evaluate our approach, we formally specified and analyzed the requirements for a subsystem of a realistic command and control system. The requirements are derived from existing command and control systems. We used the formal language and proof checker of PVS [8]. For the presentation of the literate specification we used a tool developed for literate programming, viz. `noweb` [9].

Section 2 forms the main section of this paper. A general approach for the formal specification and analysis of requirements is developed. The case study is presented in Section 3. Finally, we will evaluate our approach on the basis of the case study in Section 4. The latter also mentions some related work.

## 2 Formal requirements specification

We will distinguish two activities: specification and analysis of the requirements. These activities interact in an iterative process, such that analysis leads to a specification of enhanced quality.

The specification of the requirements consists of building three views, or models: the information-, function- and control-model. The information model describes the static aspects of the system, whereas the function- and control model describe the behavior. In the function model, the various operations on the state are defined. The control model specifies the actual interaction with the system, and the order in which the operations occur, in response to the input. In Section 2.2 we describe these views in more detail.

For the analysis, we distinguish between verification and validation. By *verification*, the intrinsic quality of the specification is addressed. Special attention will be given to the verification that the various views fit together. It cannot be verified however, whether the intended system has been specified, because there is no authoritative document against which the specification can be checked; this check is the purpose of the *validation* by domain experts. Section 2.3 addresses both aspects of analysis.

### 2.1 Language and form of the specification.

In order to relate the various views, there must be one underlying formalism. For this reason, we have chosen to use the same language for all models. This can be weakened by using different languages that are translated to a common underlying formal framework. The second approach is followed by [13].

This language must be expressive enough to model the various views. In order to find a lot of mistakes automatically, we prefer a language with a strong typing discipline (in the sense that every expression must have a unique type). Finally, the language must be supported by tools, in order to make the analysis of the specification practical.

We propose to develop a literate specification. This means that informal and formal requirements are interleaved. Intermediate documents also contain a list of unsolved questions. An additional advantage of a literate specification is, that it allows to circumvent the rigid order imposed by formal languages. The specification can now be presented in a top-down way.

### 2.2 Specification of the models.

**Information model.** The information model describes the static part of the specification. It can be seen as an abstraction of the *global state* of the system. In command and control systems, the information model describes the abstract picture of the environment.

Typically, the information model consists of a number of entities, together with certain relations between them. A number of constraints can restrict the set of allowed states. These constraints specify the invariant properties that the state should have. For each entity, its attributes are specified, by declaring their names and the range of values they can take. Attributes can have special properties, such as being optional.

**Function model.** The function model defines the manipulations that change the state. These manipulations can be seen as a relation between succeeding states. The manipulations carry additional arguments, representing the input to the system. Only when the system is deterministic, the manipulations can be described as functions.

**Control model.** The control model specifies the interaction of the system, for example as the set of valid sequences of atomic actions. We focus on input actions. The interaction protocol induced by the set of valid input sequences represents the assumptions about the environment.

In addition, the control model specifies how the input actions trigger the manipulations defined by the function model. In this sense, the control model defines in which order the manipulations shall be performed. It also indicates the initial state.

**Relationship between the models.** We have introduced three models: the information model, the function model and the control model. We see these models as complementary.

The function model describes possible manipulations on states. The manipulations are also constrained by the information model, because the resulting state must satisfy the constraints of the information model as well. Together these models form an abstract data type: a data structure with a set of operations.

The control model makes this abstract data type into an abstract state machine. The allowed states are defined by the information model. The accepted input is defined by the control model. The input events trigger transitions between the states, as defined by the function model.

## 2.3 Analysis.

**Verification.** We show how consistency and internal completeness can be verified. With *consistency* we mean that no contradictory requirements are given. A necessary condition is that the various views are *compatible* (they allow a common system). Another necessary condition is, that the specification is *logically consistent* (falsum is not derivable). A specification is *internally complete* if it deals with all cases that can be foreseen by inspecting the specification. This means among others that to any expected input there is some specified next state. However, we don't require that this state is deterministically specified, as opposed to [4].

First of all, it must be checked whether the specification is well-formed, which can be done by parsing and type-checking. This already reveals a lot of potential inconsistencies. In many typed formal languages, type-correct theories containing definitions only (thus excluding axioms), are even logically consistent.

The second possibility of verification is proving putative theorems [11]. These theorems can be seen as new requirements, or as challenges to the specification. The confidence in the specification is raised by proving that they already follow from the specification.

It must also be proved, that the various views are compatible, i.e. don't put contradictory requirements. This is checked by proving that a system satisfying all requirements exists. By proving that the initial state satisfies the constraints, we have a witness that the information model in isolation is consistent. To show that the information and function model are compatible, we prove that for every state (satisfying the invariants) and every input (satisfying some precondition), there exists a next state, in accordance with the requirements of both the information and the function model. In order to show that the control model is compatible with the other models, we translate this precondition into assumptions on the input. It must then be proved that the control model guarantees that these assumptions indeed hold.

Note that also internal completeness is addressed by this analysis, since for *any* valid input there is some response.

**Validation.** Validation aims at making sure that the specified system corresponds to the desired one. It is mainly the task of the domain experts. The proposed method supports validation of both informal and formal requirements, because a well-structured, literate specification is developed. Hence the information can be accessed easily.

The formal specification can be validated by inspection. This is possible because the informal requirements serve as explanation and documentation of the formal specification. This part of the validation ensures that the formalization has been carried out correctly.

The validation of the informal requirements is supported too. The central point here is the questionnaire. The attempt to formalize requirements forces one to address ambiguities and incompleteness. This usually requires additional information from the domain experts. A second source of questions is the formal verification of the specification. Failure of type-checking and non-provability of putative theorems often indicate errors.

According to the answers to these questions, the informal and/or formal requirements are updated. This leads to an improved version of the requirements specification.

# 3 Specification and analysis of track joining

We illustrate our method for formal specification by a case study. As an example, we selected a component of command and control systems, related to track joining. We first briefly present the general problem statement of track joining (3.1). Then we describe the language and tools that were used (3.2). The specification and analysis of the requirements along the lines of Section 2 is presented in Sections 3.3–3.6.

Using our method, the specification was derived from existing (informal) requirements specifications for track joining. Below we cite from the literate specification that we developed. Rather than presenting a polished final result, we report on an intermediate stage and show how analysis can lead to an improved specification.

### 3.1 Automatic track joining

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

- Sensor tracks, as reported by a sensor.
- Tactical tracks, as presented to the operator.

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, various sensor tracks should be *joined* into a single *tactical* track, if they represent the same real-world object. One of the tasks of Tactical Track Management (TTM) is to derive and maintain the set of tactical tracks.

Precisely one of the sensor tracks that are joined to a particular tactical track (viz. that with the highest accuracy) is *responsible* for reporting on the current position, velocity and other attributes of the tactical track.

The various sensors can initiate new sensor tracks, and update or wipe existing sensor tracks; this is the input to the system. TTM has no output actions. Instead, an abstract picture is built, containing the current sets of sensor- and tactical tracks, and the join- and responsibility relations between them. A similar assumption on systems is made in [13].

### 3.2 PVS and noweb

In order to carry out the case study, a particular formal language has to be chosen. To support the case study mechanically, certain tools must be present. We used PVS (Prototype Verification System) [8] as a specification language equipped with an interactive proof checker, and `noweb` [9] as a literate specification tool.

**PVS language.** The language of PVS is based on classical higher-order logic. This means that quantification over functions, sets and properties is allowed, leading to a great expressive power. The logic is equipped with a type system. PVS has a rich variety of types, e.g. numeric types (`nat` and `real`), enumerated types ({`red,white,blue`}), pairs (`[nat,bool]`), functions (`[real,nat->real]`), subtypes ({`x:nat| p(x)`}), record types (`[# name:string, age:nat #]`) and a scheme for defining abstract data types, such as (recursive) trees.

A large library contains definitions of many general concepts, like lists, sequences, induction, etc.

**PVS system.** A specification is parsed and type-checked by the PVS system. Due to the typing rules (especially subtyping), it is undecidable whether a theory is type-correct. To overcome this, the type-checker generates type-check conditions (TCCs) that are sufficient for type-correctness.

Theorems raised by type-checking or by the user can be interactively proved in the proof checker. The user repeatedly applies the rules of higher-order logic, in order to simplify the goal to be proved, until it is trivial. This process is partly automated by built-in strategies, like term rewriting, and decision procedures for linear arithmetic and propositional logic.

**Noweb.** We use `noweb` as a literate specification tool [9]. In combination with LaTeX, `noweb` yields typeset text and PVS code. The PVS code can also be extracted, in order to formally analyze it.

PVS theories are split in chunks, which can be presented in any order. These chunks have labels and can refer to each other. In the case study presented in the next sections, text in `type-writer` font is PVS code. The italic parts between angled brackets are ⟨⟨*labels*⟩⟩.

### 3.3 Information model

The information model describes the static structure and the invariant properties of the track database. Using conventional methods, the informal specification can be expressed by an ER-diagram as depicted in Figure 1.

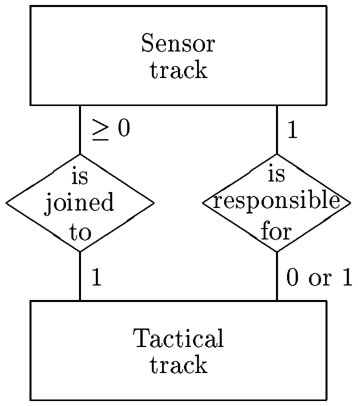

Figure 1: ER-diagram for track joining

The diagram introduces two kinds of entities (sensor and tactical tracks) and two relations between them (the join- and responsibility-relation). The numbers on the edges indicate cardinality constraints on the

```
⟨⟨tdb.pvs⟩⟩ ≡
track_database: THEORY
BEGIN
  ⟨⟨Track definitions⟩⟩
  Data_structure: TYPE =
  [# s_tracks: setof[Sens_track],
     t_tracks: setof[Tac_track],
     join: pred[[Sens_track,Tac_track]],
     resp: pred[[Sens_track,Tac_track]]
  #]
  X,Y: VAR Data_structure
  ⟨⟨Constraint definitions⟩⟩
  TDB: TYPE = {X | constraints(X)}
END track_database
```

Figure 2: Topmost specification of track database

```
⟨⟨Constraint definitions⟩⟩ ≡
s,s1,s2: VAR Sens_track
t: VAR Tac_track

constraint1(X):bool = FORALL s,t:
  resp(X)(s,t) => join(X)(s,t)
constraint2(X):bool = FORALL s:
  s_tracks(X)(s) => exists1! t: join(X)(s,t)
constraint3(X):bool = FORALL s1,s2:
  s_tracks(X)(s1) & s_tracks(X)(s2) &
  number(s1)=number(s2) => s1=s2
...
constraints(X):bool =
   constraint1(X) & constraint2(X) & ...
```

Figure 3: Constraint definitions

relations, e.g. a sensor track is responsible for at most one tactical track.

Other types of constraints are usually stated in accompanying text, like the constraint that a sensor track can only be responsible for a tactical track if it is joined to it. Typically, a data dictionary defines the attributes of the entities, and the types of the various attributes.

**Formalization of the track database.** The track database is formally specified in two steps. First we define a data structure, which roughly indicates what the database looks like. Then the constraints are defined as predicates on this data structure. A track database (type `TDB`) is then defined as a data structure that satisfies the constraints.

The topmost part of the formalization is displayed in Figure 2. This part corresponds to the ER-diagram without the constraints. The data structure is defined, as a record containing sets of sensor and tactical tracks and the responsibility- and join-relations between them (represented as predicates on pairs).

The conjunction of the constraints forms a predicate on the data structure. In Figure 3 we present some of the constraints. The first constraint expresses that a responsible primitive track must be joined to the corresponding tactical track. The second one expresses a cardinality constraint in the ER-diagram in Figure 1: every sensor track is joined to exactly one (indicated by `exists1!`) tactical track. Finally, constraint three states that track numbers are unique.

**Entities.** Figure 4 illustrates the definition of sensor and tactical tracks. The types and the attributes of the tracks are defined, respectively. Note that we declared `identity` as *optional*. The type `optional[t]` is an

```
⟨⟨Track definitions⟩⟩ ≡
Source: TYPE+
Sens_number: TYPE+
Identity: TYPE =
  {friend, hostile, pending, joker, faker}
Kinetic: TYPE = [# px,py,vx,vy: real #]
Max_accuracy: nat
Accuracy_level: TYPE =
  {x:nat | 0 <= x & x <= Max_accuracy}
Sens_track: TYPE =
[# number: Sens_number,
   source: Source,
   identity: optional[Identity],
   kinetics: Kinetic,
   accuracy: Accuracy_level
#]
Tac_track: TYPE = ...
```

Figure 4: Types and attributes of tracks

abstract data type with type parameter `t`; we omitted its definition. The elements of `optional[t]` are either `none` (absent) or `one(x)`, where `x` is of type `t`.

## 3.4 Function model

The function model introduces the manipulations that change the track database. The external manipulations consist of creating, updating and wiping sensor tracks. Only a cross-section of the specification of the manipulations will be given. As an illustration we consider the requirements related to the creation of a new sensor track into the track database. Requirements for the other external manipulations are defined similarly.

Informally, the requirements can be stated as fol-

```
manipulations: THEORY
BEGIN
IMPORTING track_database
⟨⟨definition of distance_crit⟩⟩
⟨⟨definition of determine_resp⟩⟩
⟨⟨definition of new_tac_track⟩⟩

no_conflict(id1,id2:Identity):bool= TABLE
  id1, id2 |[pending| friend |hostile]|
%-------------------------------------
|pending   | TRUE   |  TRUE  | TRUE  ||
|friend    | TRUE   |  TRUE  | FALSE ||
|hostile   | TRUE   |  FALSE | TRUE  ||
ENDTABLE%----------------------------

join_criteria(s,t):bool =
  distance_crit(kinetics(s),kinetics(t))
& no_conflict(identity(s),identity(t))

join_sens_track(s)(X,Y): bool =
EXISTS t:
  t_tracks(X)(t)
& join_criteria(s,t)
& LET Z=X WITH [join:=add((s,t),join(X))]
  IN determine_resp(t)(Z,Y)

new_sens_track(s)(X,Y):bool =
LET Z=X WITH [s_tracks:=add(s,s_tracks(X))]
  IN    join_sens_track(s)(Z,Y)
     OR new_tac_track(s)(Z,Y)
END manipulations
```

Figure 5: Part of the function model.

lows: If a new sensor track is inserted, it shall be verified whether it can be joined to an existing tactical track. This must be done according to two criteria:

- the distance between the two tracks is within a specified margin;
- the identity of the tracks is not conflicting (as defined by the table in Figure 5).

If a tactical track satisfying these criteria exists, the new sensor track is automatically joined to it, and the responsibility relation is updated accordingly. Otherwise a new tactical track must be created.

Note that this description refers to internal manipulations, like joining a sensor track to a tactical track, creating a new tactical track, and determining the responsible track. The various manipulations can be formalized in a rather straightforward manner. Part of the formal function model is illustrated in Figure 5.

In the definition of `new_sens_track(s)`, `s` is added to the sensor tracks in state `X`, resulting in an intermediate state `Z`. Note that `s` has not been joined to a tactical track, so `Z` doesn't satisfy `constraint2`. This is repaired in the final state `Y`, by either joining `s` to some existing tactical track, or by creating a new tactical track and joining `s` to it (the specification of the latter manipulation is not shown here).

The relationship `join_sens_track(s)(X,Y)` holds if sensor track `s` is joined to some existing tactical track `t` satisfying the join criteria. In that case, the pair `(s,t)` is added to the join relation, and the responsible track is re-determined (which is not further specified in this paper). Whether identities are non-conflicting is defined by a table, a PVS construct that is tested for completeness automatically, as explained in Section 3.6.

## 3.5 Control model

The task of the control model is to define the interaction between the (sub)system and its environment. In this case study, this boils down to defining the valid sequences of input actions. We first define the atomic `actions` as an abstract data type, with elements of the form `new(s)`, `update(s)` and `wipe(s)`, for some sensor track `s`. In all cases, the function `arg` returns the sensor track `s`. The PVS theory containing the control model starts with the following declarations:

```
i,j,k: VAR nat
input: VAR sequence[actions]
```

The `input` to TTM is represented by a sequence of actions. Thus `input(i)` denotes the $\mathtt{i}^{\mathrm{th}}$ action, and `arg(input(i))` the corresponding sensor track.

A predicate `present(input)(s,i):bool` (sensor track `s` exists after `i` actions of the `input` have occurred) can be defined as follows: At some point `j<i`, a track with `s`'s number is initiated and it is not wiped for any `j<k<i`. The predicate `valid(input):bool` holds if all newly initiated tracks are not present, but all updated and wiped tracks are.

The control model is linked to the information and function model by defining an initial state and a next-state relation. The `initial_state:TDB` contains an empty set of sensor- and tactical tracks. The higher-order function `next_state`, mapping actions to the corresponding relations in the function model, is defined such that e.g. `next_state(new(s))` = `new_sens_track(s)`. Using these definitions, we can define `state(input)(i)(X):bool`, which is true of a state `X` if it is reachable from the initial state by performing the first `i` `input` actions.

### 3.6 Verification

**Type-checking.** The first formal check on the specification concerns type-correctness. PVS detects a type error in the theory representing the function model (cf. Figure 5): `no_conflict` expects arguments of type `Identity`, but in the definition of the `join_criteria` it gets arguments of type `optional[Identity]`. This leads to a question for the domain experts, how the `no_conflict`-criterion is to be interpreted in case identities are absent.

Recall that type-checking may introduce type-check conditions. We show one of the TCCs automatically generated by PVS, that needs careful consideration:

```
no_conflict_TCC: OBLIGATION FORALL id1, id2:
      id1 = pending => NOT joker?(id2)
```

This TCC reveals an omission in the specification. It is due to the fact that the `TABLE` defining the predicate `no_conflict` (cf. Figure 5) is not exhaustive: the rows and columns of `joker` and `faker` are missing. PVS wants us to prove that these cases never occur. This obligation cannot be fulfilled; the specification has to be amended by extending the table.

**Consistency between views.** We now show how consistency between the various views can be stated and proved. From now on, `X,Y` only range over `TDB`, i.e. data structures satisfying the constraints.

```
X,Y: VAR TDB
```

As indicated in Section 2.3, compatibility of a manipulation with the information model is stated by the formula `FORALL X,s:  EXISTS Y: R(s)(X,Y)`. This formula expresses that in any state `X` satisfying the constraints, the transition `R` with input `s` can be performed leading to some state `Y` satisfying the constraints. However, this doesn't hold for the manipulation `new_sens_track`: adding a track with an existing `number` violates `constraint3`, which expresses that numbers are unique. We are forced to think about the necessary precondition for this manipulation. We now get the following, which can be proved with some effort in the PVS proof checker. The lemmata and theorems below are translated to their universal closure implicitly.

```
precond(X,s):bool =
  NOT EXISTS s1:
    s_tracks(X)(s1) & number(s)=number(s1)

info_fun_compatible: LEMMA precond(X,s) =>
  EXISTS Y: new_sens_track(s)(X,Y)
```

Of course, it must also be shown that assuming the precondition is justified. This is done by proving that given `valid` input (in the sense of the control model), the assumption can be proved. Formally, this is stated as follows (If after `i` actions of `valid input` we get a `new(s)` action in state `X`, then the `precondition` holds):

```
precond_justified: LEMMA valid(input) &
    new?(input(i)) & state(input)(i)(X)
 => precond(X,arg(input(i)))
```

After doing the same analysis for the other manipulations, we can prove that for valid input, the system is always in some specified state. The theorem below can be proved by induction on `i`. The base case holds since the initial configuration satisfies the constraints. The induction step uses that the manipulations are compatible with the information model.

```
views_compatible: THEOREM  valid(input) =>
  EXISTS X: state(input)(i)(X)
```

## 4 Concluding remarks

**Evaluation.** The case study showed us that our approach has certain benefits, but also that certain improvements are still needed. We mention some benefits of our approach:

- Already the attempt to formalize brought about a lot of good questions and conceptual clearness.
- Type checking automatically found forgotten cases in the definition of tables, functions, etc.
- The control model forced us to give a clear boundary of the system. Note that the function model doesn't make clear that `join_sens_track` is an externally invisible manipulation.
- The invariants of the information model are a means to control the system integrity. In the case study for instance, moving the `join_criteria` towards the information model would imply that they always hold, not only when the join-manipulation is performed.
- The verification of compatibility between views forced us to think about the assumptions made on the environment. This led to the introduction of preconditions.

Note that a formal analysis not only uncovers various errors early, but in the end we have a *guarantee* that the specification is non-ambiguous, internally complete, and logical consistent and that the various views are compatible.

The case study also revealed that certain improvements are needed:

- Technically, the combination of PVS and `noweb` is inconvenient. The PVS system works on the extracted code, so feedback (e.g. type errors) is directed to this code, instead of to the source file. Similarly, since `noweb` generates the output, it is not possible to use the LaTeX pretty printer of PVS. Also graphical notations, like OMT-diagrams [10], should be supported by the literate specification tool.

- More emphasis should be put on the control model. Also output actions have to modeled, and sequences of actions are not expressive enough in general, although they were sufficient in this case study.
- A clearer view is needed on the exact tests that should be performed in the analysis. This should result in a clearer description of *internal completeness* and *consistency.*

**Future work.** Eventually, our research should lead to a system-wide formal specification of command and control systems, resulting in formal development. The following issues still have to be addressed:

- *System-wide specification.* Other typical subsystems of command and control systems must be tried, e.g. the execution of corrective actions. Moreover, special attention must be paid to the composition of various components. Until now only views of the same component are supported.
- *Non-functional requirements.* Important issues, like real-time requirements, graceful degradation, robustness and availability of the system are not yet dealt with.
- *Optimization Criteria.* Optimization criteria occur frequently in command and control systems. It is not easy to formalize that a system should use/satisfy such criteria. The criteria themselves can be stated of course, but note that in general, we cannot *require* that a system finds the optimal solution. Similarly, as signaled by [3], mere preferences (as opposed to requirements) are hard to formalize.

**Related work.** We refer to [1] for an overview of industrial applications of formal methods. In the papers below, similar work is reported as in our paper.

A requirements specification for an aircraft collision avoidance system is given in [6]. A single state-based model is constructed in that paper, written in RSML (Requirements State Machine Language). This model expresses the black-box behavior of the system. Several formal properties of the specification are automatically verified in [4].

In [3], formal methods are used for specification, design and verification of an air traffic control information system. In that paper, ER-diagrams and dataflow diagrams are translated into state- and operation-specifications in VDM [5]. Concurrency requirements are modeled separately. We share many of the author's findings concerning specification.

A formal requirements analysis for an avionics control system can be found in [2]. PVS is used there to formulate functional and safety requirements. It is formally verified whether the functional requirements satisfy the safety requirements.

**Acknowledgments.** We like to thank Dieter Hammer for his stimulating contribution to our project discussions and his comments on this paper.

## References

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

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

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

[4] 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.

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

[6] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese. Requirements specification for process-control systems. *IEEE Trans. on SE*, 20(9):684–707, 1994.

[7] B. Meyer. The next software breakthrough. *IEEE Computer*, 30(7):113–114, 1997.

[8] 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.

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

[10] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, 1991.

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

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

[13] P. Zave and M. Jackson. Where do operations come from? A multiparadigm specification technique. *IEEE Trans. on SE*, 22(7):508–528, 1996.