The Engineering Roles of Requirements and Specification

Elisabeth A. Strunk[1], Carlo A. Furia[2], Matteo Rossi[2], John C. Knight[1], Dino Mandrioli[2]

[1] University of Virginia
151 Engineer's Way
Charlottesville, VA 22904-4740
{*strunk, knight* }*@cs.virginia.edu*

[2] Politecnico di Milano
Via Ponzio 34/5,
20133, Milano, Italy
{*furia, mandrioli, rossi*}*@elet.polimi.it*

Abstract

The distinction between requirements and specification is often confused in practice. This obstructs the system validation process, because it is unclear what exactly should be validated, and against what it should be validated. The reference model of Gunter et al. addresses this difficulty by providing a framework within which requirements can be distinguished from specification. It separates world phenomena from machine phenomena. However, it does not explain how the characterization can be used to help assure system validity.

In this paper, we enhance the reference model to account for certain key elements that are necessary to expose and clarify the distinction and the link between requirements and specification. We use the enhanced version to present a more refined picture of validity, where validation has two steps that can be undertaken separately. We use this picture to question whether the "what the system will do, not how it will do it" paradigm is useful in describing how to construct a specification, and propose an alternative. Finally, we present the requirements and specification for an illustrative example based on a runway incursion prevention system, with the ArchiTRIO formal language in a UML-like environment, to show how this might be done in practice.

1 Introduction

The earliest phases of the software lifecycle exert a large influence on software cost and quality, yet they are the least amenable to analysis and tend to be poorly understood. The distinction between requirements and specification in software engineering, for example, is often confused in practice (and even in some textbooks). This confusion translates into significant development difficulties; for example, it obstructs the system validation process because of a lack of clarity in

what exactly should be validated and against what it should be validated. In many cases, practitioners produce a document called the "Requirements Specification", where validity is considered only in an ad hoc manner, and where validity is difficult to show at all because system requirements and what amounts to system design are tightly coupled.

In a similar way, validation and verification are often confused. This confusion occurs despite the fact that the concepts are distinct and that each requires different techniques for its conduct. Making the situation worse is that there are few techniques for validation, and those that are available usually serve more than one purpose. Testing, for example, is used in practice for both validation and verification, yet which test case does which or even if these goals are separable is rarely if ever clear.

The reference model of Gunter et al. [GGJZ00] addresses these difficulties, in part, by providing a framework within which requirements are carefully distinguished from specification. It builds on earlier work by Zave and Jackson [ZJ97] that characterizes *phenomena* of interest to the system and separates world phenomena (those of the requirements) from machine phenomena (those of the specification). The reference model gives a detailed account of different classes of phenomena and the role they play in software development, but it does not explain how the characterization can be used in practice to support software development—for example, to help assure system validity.

In this paper, we introduce an enhanced reference model in which the distinction between requirements and specification is further refined and clarified. The refinement of the model enables us to construct a two-stage validation process, where each stage addresses a specific aspect of validity. We show how our validation process can be carried out with three different techniques, depending on how formal someone using the reference model chooses to be in creating system artifacts.

Finally, we use this model to question whether the "what the system will do, not how it will do it" [IEEE98, Sec. 4.2] paradigm is a useful one in describing how to construct a specification. In the case of the argument presented here, *requirements* set out the effect the system must have on its environment and the *specification* details the functionality necessary to achieve the requirements. Thus, the specification of a system is a form of high-level design that has a specific role, similar to a software architecture that forms the design layer underneath the specification.

These ideas are not purely theoretical, and in order to demonstrate their utility, we present the requirements and specification we developed for an illustrative example based on the Runway Safety Monitor (RSM) [Gre00], part of a system to prevent airport runway incursions, to show how our approach might be applied in practice. We constructed both the requirements and specification for the RSM, using the ArchiTRIO formal language [PRM05] in a UML-like environment. We describe the various artifacts produced and our experience producing them, giving concrete examples of artifacts and variables in the different categories of our enhanced reference model.

The remainder of this paper is organized as follows. Section 2 explains the difference between requirements and specification, reviewing the Reference Model and its classes of phenomena. Section 3 shows how the enhanced Reference Model clarifies the objectives of the validation process, while Section 4 questions the traditional perspective on "requirements vs. specification" in accordance with the enhancement to the Reference Model. Section 5 briefly introduces the TRIO formal language, and its UML-compatible derivative ArchiTRIO. Section 6 presents the RSM illustrative example, and Sections 7 and 8 respectively describe how the RSM formal requirements and specification were constructed in our framework. Finally, Section 9 concludes.

2 Requirements vs. Specification

2.1 Distinguishing Between the World and the Machine

The fundamental idea underlying Gunter et al.'s reference model [GGJZ00], and the earlier work on which it is based [ZJ97], is that entities (*phenomena*) in a system's requirements (*world* phenomena) are distinct from *machine* phenomena. This distinction is not an obvious one to many software professionals, because developers are used to working with a model of a system and substituting the model for reality. To clarify the distinction, we present an example based on the simple notion of an aircraft's altitude. An aircraft's altitude is a world phenomenon. The way that a computing system will be able to compute useful results using the aircraft's altitude, however, is through its sensors, which are part of its control system. Aircraft software performs computations on the sensor reading (a machine phenomenon), *not* the actual altitude, because the software is not able to know the true altitude.

Generally, the sensor readings are precise enough that substituting them for the actual altitude is sufficient. There are, nevertheless, two reasons why it is important to distinguish between the world value and the machine value in general. First, the world value might not be controllable or measurable directly. In this case, the way it is controlled or measured is part of the design of the system, not its requirements. Domain experts should not be saddled with system design when developing requirements or assessing system validity, because this adds complexity to an already difficult job.

The second reason why the distinction between world and machine variables is important is that the machine value can accrue error when combined with other machine values. The error for individual variables might be simple enough for a person to deal with unaided in constructing requirements, but the error composition can be more complex and might need analytical techniques for its computation. It is important that such differences be documented accurately and systematically.

2.2 The Reference Model

Gunter et al. [GGJZ00] present a reference model for distinguishing between classes of requirements artifacts and specification artifacts. Their model has five major classes of artifacts, as shown in Figure 1 (our figure differs in appearance from that presented by Gunter et al., although the content is identical). The artifacts are broken into two types: those representing the environment, and those representing the system.

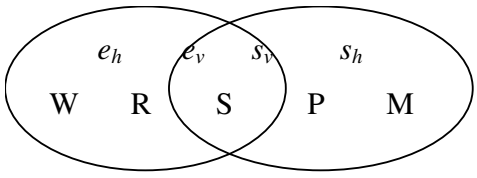

Figure 1 - Gunter et al.'s Reference Model

- W is the set of artifacts that describe assumptions that can be made about the environment.
- R is the set of artifacts that describe what the system is required to make true of the environment.
- P is the software program.
- M is the machine on which the program operates.

- S represents the specification which, as shown in the diagram, resides between the system and the environment.

Within these artifacts, there are phenomena that belong to (and are controlled by) the environment (designated by the set *e*) and phenomena that belong to (and are controlled by) the system (designated by the set *s*). The term *environment* represents a broader view of the term *world*, where the world has been augmented with the requirements of the system that will be built. The term *system* represents a broader view of *machine*, where the system can also include a model of the machine's interaction with its environment. The phenomena that are visible by both the environment and the system—and thus shared between the two—are designated by the subscript *v*, and those that are seen only by one or the other are designated by the subscript *h*. Thus, for example, the environmental variables that are visible to the system are designated as e_v.

The reference model of Gunter et al. is an excellent starting point for discussion of components of system development, but still leaves certain concepts slightly fuzzy. They state that the specification crosses the border between the environment and the system, since it is allowed to reference both visible machine variables and visible environment variables. We disagree and maintain that the specification is stated purely in terms of machine variables. The key to understanding our use of and *extensions* to the reference model is in seeing the difference between requirements and specification, as we explain in the next section.

2.3 Formalizing Requirements

Distinguishing clearly and explicitly between the world and the machine means that requirements and specification include two separate models that play two separate roles. Because in the past the distinction has been blurred, many people think of requirements as an informal concept and specification as a formal one. This categorization, however, is an incidental one. R (in the model) can be formalized just as well as S can; it will simply use different variables representing different phenomena. It is also common practice to think of S's variables as representing R's phenomena, but that leads to the use of S's variables as entities in computations. This means that S's variables truly belong to S, and their link to R's phenomena is actually never stated! If R's phenomena, on the other hand, are represented formally, then the relationship of S's variables to R's phenomena can be stated formally as well.

In terms of our earlier example of an aircraft's altitude, imagine that in R is a requirement that an aircraft's pilot be alerted if the aircraft's altitude falls below a threshold that is defined to be safe. R now has three phenomena: *r_alt*, the actual altitude of the aircraft; r_alt_{safe}, the minimum safe altitude of the aircraft; and *r_alert*, whether the system has issued an alert to the pilot. The requirement is now:

$$r\_alt < r\_alt_{safe} \;\Rightarrow\; r\_alert.$$

The specification might say that if the measured altitude falls below the minimum safe altitude, then an alert is raised. The specification, then, also has three machine phenomena: *s_alt*, the aircraft's measured altitude; s_alt_{safe}, the constant in the program below which the measured altitude may not fall; and *s_alert*, whether the system has output a command for an alert to be raised. The specification is now:

$$s\_alt < s\_alt_{safe} \;\Rightarrow\; s\_alert$$

With the requirement and specification both in place, we can determine whether the specification satisfies its requirement. Given our knowledge of the domain, we will make three assumptions. In the first, we assume that the measured altitude is within three meters of the actual altitude:

$$r\_alt - 3 \leq s\_alt$$

and

$$s\_alt \leq r\_alt + 3$$

In the second, we assume that if the system signals an alert, an alert is raised:

$$r\_alert = s\_alert$$

and in the third we assume that the constant s_alt_{safe} is the same as r_alt_{safe}

$$s\_alt_{safe} = r\_alt_{safe}$$

The observant reader will have noted the mistake in the specification. Because of the error present in the altitude measurement, we cannot know precisely when the aircraft passes the minimum threshold. We must, instead, change s_alt_{safe} to be equal to $r\_alt_{safe} + 3$ in order to ensure that the requirement is satisfied. While this mistake might seem trivial, such mistakes have the potential for causing failure in any system. Indeed, many embedded systems have a much smaller margin of safety than is suggested by our simple example.

In current practice, people claim that the goal of a specification is to enable developers and domain experts to state *what* the system must do at a high level of abstraction, giving them support for validating the specified system operations without bringing in design and other development decisions. The problem with this view is that within it specifications typically are a mix of requirements written using world variables and design written using machine variables, with no clear delineation or relationship between the two sets.

Thinking of requirements as an entity that is entirely separate from the specification introduces the possibility of achieving the stated goal of the specification, because it separates the world variables (and thus the requirements) from the machine variables (the design component of the specification). Consider, for instance, the following change to our altitude example. If instead of raising an alert we specified that control surfaces be altered automatically to force the aircraft to gain altitude (as might occur with an autopilot), we could change the requirement to the invariant:

$$r\_alt > r\_alt_{safe.}$$

The control algorithms of the aircraft would then have to be related to the physical altitude of the aircraft through control equations if we are to claim that the specification implemented its requirement. Those algorithms would not, however, need to be considered by the requirements analyst—which they are in many cases today.

2.4 The Enhanced Reference Model

In order to support the strict separation of requirements and specification that we seek, we make two fundamental changes to the model of Gunter et al. The enhanced model that we propose is shown in Figure 2.

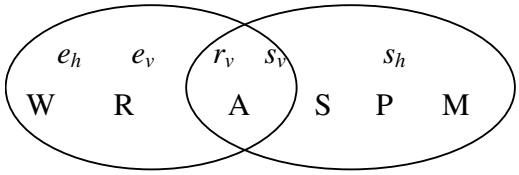

Figure 2 - Enhanced Reference Model

First, we do not include any variable which is shared *a priori* between the environment and the system: the relationship between visible environment and system variables has to be stated

explicitly. Thus, we do not allow S to use any variables from the environment. The original reference model assumes that environment visible values can be referenced directly in the specification, which corresponds to assuming that they can be measured exactly. This direct use of the environment values also implies that the specified artifact can monitor or affect directly the environment. Instead, we place a middle layer between S and R, enabling us to cleanly and explicitly separate the actual values from measured values, but still be able to speak about their relationship. We call this layer the set A. A is a set of axioms, as seen from the specification's perspective, or assumptions, as seen from the requirements' perspective. It relates symbols in the formal specification model to symbols in the formal requirements. In our altitude example above, the set A is:

$$\begin{array}{lcl} \{\ r\_alt - 3 & \leq & s\_alt \\ s\_alt & \leq & r\_alt + 3 \\ r\_alert & = & s\_alert, \\ s\_alt_{safe} & = & r\_alt_{safe}\ \}. \end{array}$$

A is similar to the idea of domain knowledge, represented by K in the previous work of Zave and Jackson [ZJ97]. We have introduced a separate set because domain knowledge typically encompasses much more than the formal relationships between variables in R and variables in S that we need to link the two formalisms. A is, in essence, a way of encapsulating the formal relationships between the phenomena in R and those in S. While we could define the mapping between these phenomena to be a trivial one (i.e., particular phenomena in R are essentially equivalent to particular phenomena in S), as was implicitly done in the original reference model, we feel that this is not the most effective approach to take in the engineering of large systems. The main reason lies in the environment: with the ability to introduce a complex mapping in A, environmental phenomena used in the requirements no longer need be restricted to those in e_v. Instead, the relationship between e_h and s_v can be set out in A. This advantage is discussed in more detail below.

The second fundamental change that we make to the reference model is that we distinguish between the *model* of the environment used in the requirements and the *requirements* themselves. Requirements are conceptual entities created in a customer's mind; they are fundamentally informal, and there is no way to be sure that any representation of them will

convey the desired concept to others completely and correctly. When those requirements are enunciated, a distinction is created between the concepts of the user and the information that is described to the developer or set out in the requirements document and the concepts that exist in the mind of the customer. We call the enunciated concepts the *model* of the requirements. For instance, *altitude* is a concept, but *r_alt* might be a value corresponding to an aircraft's altitude above ground level, as measured in meters. *s_alt* would then be the system's measurement of the aircraft's true altitude above ground level in meters. If the requirements are not represented formally, then the model is the informal information conveyed in whatever form the system's statement of requirements takes.

The distinction between the requirements and their model essentially defines the classic validation problem: whether what the system does—which is defined by the logic of its software—is what was intended of it. To make this distinction, we introduce in the enhanced reference model a new set of phenomena: r_v. r_v is the documented model of the phenomena in e_v. In the example above, *altitude* is a member of e_v, *r_alt* is a member of r_v, and *s_alt* is a member of s_v. We make this distinction to support validation, as explained in Section 3.

2.5 The Four-Variable Model and Related Work

Some aspects of our enhanced reference model are similar to Parnas and Madey's *four-variable model* for documenting computing systems [PM95], and extensions thereof [MT01]. Both our work and Parnas and Madey's set out a precise distinction between different documents of a developed system. More precisely, Parnas and Madey's notion of *Software Behavior Specification*, represented by a relation named SOF, is akin our notion of specification, as they both refer to machine variables only. Also, the two relations IN and OUT in the four-variable model constitute an explicit link between input/output (machine) variables and environment variables, and thus are analogous to our notion of set A. Notice that IN and OUT are said to document system design in [PM95].

There are four main differences between the four-variable model and our enhanced reference model. First, whereas Parnas and Madey's model is presented in a mostly theoretical way, relying on the existence of mathematical relations between variables—which may be too restrictive in practice—our model presents unambiguously a clear distinction between requirements and specification without relying *a priori* on any formal assumption. As Gunter et

al. note in their paper [GGJZ00], Parnas and Madey's model can be regarded as a specific instance of the more general reference model.

Second, our enhanced reference model makes a distinction between the environment and the formal model of the environment. NAT and REQ—in the four-variable model—are both formal relations; they are unable to express informal requirements concepts and so do not support the two-stage validation process discussed in Section 3.

Third, our model does not force a distinction between observed and monitored environment variables, contrarily to the four-variable model. Not requiring such a distinction is more natural when dealing with very abstract descriptions such as requirements, where such a partition may still be difficult to make.

Finally, the four-variable model refers the description of all system and environment variables to an absolute real time, which is implicitly assumed to be the same throughout requirements and specification. On the contrary, our model can deal with the subtleties of the relationship between the physical real time and the "implemented" time as seen by a machine: the former is an environment variable, whereas the latter is a machine variable, so that requirements are referred to the former, while specification to the latter. Although we will not analyze the consequences of this important distinction in the present paper, they will be tackled extensively in a companion paper [FRSMK].

In addition, the present work bears some similarities with [JHJ06], in which Jackson's problem frames [Jac00] are augmented with formulas written in Duration Calculus [CHR91] to provide a mean to formalize requirements and specification of systems that are embedded in the "physical world". In particular, [JHJ06] has a notion of variables "bridging" specification and requirements; however, [JHJ06] does not make systematic use of these variables, which must in fact disappear from the requirements to make the specification implementable (see [JHJ06], Section 3.7). The present work, on the contrary, makes such variables first-class citizens of the model, and uses them to lay out in full detail the relationship between requirements and specification. Also, [JHJ06] deals to some extent with the problem of time approximation and uncertainty, which we treat in Section 8 of the present paper (and also in greater depth in the companion paper [FRSMK]).

Hall and Rapanotti have also extended the original reference model, with mechanisms to capture and analyze system behavior over time [HR03]. While a major focus of this paper is on

timing properties, they are real-time system properties expressed within the reference model rather than overall system timing properties and so are different from the class of timing properties addressed by Hall and Rapanotti's work. Their work, and the proof obligations that accompany their analysis, could be applied to our system. Our focus, however, was on basic requirements and specification concerns in production systems, and so we leave application of their more complex analyses to future work.

We notice that the original reference model has been the object of other works aiming at providing a way to make the validation process systematic and properly integrated with the other phases of software development. Among others, we note the work by Hall et al. [HRJ05] with reference to Jackson's problem frames notation [Jac00].

3 A New Perspective on Validation

3.1 The Two Parts of Validation

A significant advantage in practice of this viewpoint on requirements and specification appears when validating a system. Traditionally, validation is the process of deciding whether a software system does what its customer wants it to do. This is a very vague notion, and it is therefore hard to achieve. While clarifying the notion does not mean it is now easy to achieve, it does convey a more complete structure for thinking about how to tackle the problem.

With the enhanced reference model, validation can now be split into two parts. The first part is matching the formal requirements model, expressed using the variables in r_v, to the informal customer desires that are formulated using phenomena in e_v and e_h. This is the most difficult part of validation because: (1) it can only be done informally; and (2) often the customer does not have a particularly clear picture of what he or she truly wants. In our example above, the user might not have remembered to document that altitude is measured from ground level, and the developer might have *assumed* that all altitudes in the domain are measured relative to the mean sea level of the Earth. This part of the validation problem is made easier, however, by limiting the phenomena of interest to only those present in the environment. As explained above, in a control system for example, the system's goal can be stated independently of the algorithms used to achieve the goal.

The second part of validation is to ensure that a specification satisfies its requirements. In our formal requirements example, we showed how to construct a specification when the requirements are simple to implement and the specification can follow directly from the requirements. In more complex systems, the specification will be a very high-level system design; in a control system, for example, the specification would be the control algorithms to be implemented.

By separating the two parts, we can isolate the second component of validation and complete it in whatever manner makes most sense for the system—but *separate* from the original requirements elicitation and validation process. We explained above how this worked for our altitude example, and we explain next how other techniques could be used to accomplish this.

3.2 Completing the Validation Process

The enhanced reference model enables a variety of techniques to be used to complete the second part of the validation process. Here we briefly review three techniques, which are not mutually exclusive and can be applied in practice according to different "recipes".

3.2.1 Formal Verification

If both requirements and specification are formalized, then the second part of validation can be based on proof. This proof, although similar to that which is used in formal verification, might involve formalisms from the application domain. For example, in a control system the proof that a particular control algorithm in the specification meets a control property stated in the requirements might require analysis from the field of control theory.

3.2.2 Informal Argument

Assurance can also be based on a rigorous but informal argument. To be effective, the argument would have to be complete, consistent, and accessible, and a candidate technology for constructing and documenting such arguments is the *assurance case*. A *safety case* is a specific type of assurance case that is used frequently to construct arguments about safety. An assurance case is a combination of a goal that has to be achieved and evidence, assumptions, context and argument strategies designed to convince the reader that the goal has been met. If the system requirements are stated as the goal, then an assurance case can be constructed from the details of the specification.

3.2.3 Execution Time Monitoring

The notion of monitoring certain desired properties of a system—or assumed properties of the environment—during operation and taking action if one or more is about to be violated has been advocated by researchers and applied by practitioners. Safety properties are an example, and this has led to the notion of a monitoring structure called a *safety kernel* [Rus89]. The safety kernel monitors the system for potential violations of safety polices and intercedes if a violation is about to occur.

In practice, the properties of interest are, in fact, requirements and what a structure like a safety kernel is implementing is execution-time assurance that the implementation meets the requirements, i.e., a combination of the second part of validation and verification of the implementation. Thus a third approach to the second part of validation is to map the requirements into a specification that literally says "the requirements have to hold", implement that specification, and execute it in parallel with the system with authority to act in some prescribed way if the system might violate some aspect of the requirements.

4 A New Perspective on "What vs. How"

Software engineers routinely define a specification as "what the system will do, not how it will do it." [IEEE98] The *how* is said to be the job of an implementation.

Jackson has observed that this distinction is not as useful as it is claimed to be [Jac00]. He says that, rather than ask *what* or *how*, people should be asking *where*. He answers the question of *where* with either *problem* or *solution*. *Problem* loosely relates to *requirements* and *solution* loosely relates to *specification*, but the relationship is not as distinct as we would like. We answer the question of *where* with either *requirements* or *specification* directly, so that it can be used to help define the scope of the two documents.

We believe that any departure from pure requirements is, in some sense, a part of the *how*. For example, setting s_alt_{safe} equal to r_alt_{safe} + 3 (in our altitude example above), is a design decision made to enable the system to satisfy its requirement in terms of *r_alt*. That design decision was made because of another design decision involving real-time aspects of the system's sensing components. From this perspective, the specification is actually a very high level design of how the system will implement its requirements. Taking this perspective does not

mean that the term *specification* is not a useful one; however, we need to look in more detail to find the most useful engineering interpretation of the word.

In this paper, we present a rigorous definition of the term specification: a specification is a very abstract representation of a system expressed solely in variables that are visible to the system. We have briefly explained, and will elaborate below, how a specification can be derived from the associated system requirements.

Our definition has two advantages over previous definitions. First, it provides a clear distinction between requirements and specification. This means that it is possible to determine whether a given entity is a specification: a human (in the case of informal documents) or a static analyzer (in the case of formal documents) can tell whether the variables used in a specification are distinct from the variables used in the requirements. Current definitions provide no clear way to determine whether a document is a specification or merely a set of requirements. This definition of specification, along with the separation of a specification from its requirements, will help industrial developers understand the role of each document, even if the requirements are not formalized.

The second major advantage of our definition of the terms, combined with our enhancement to the reference model, is the ability to push the requirements further in the world by adding additional domain knowledge. The original reference model states that the specification may not include variables used in e_h. This is sensible in that model, since the system cannot measure values in e_h. The requirements, however, may include members of e_h; and, in fact, it is possible to expand our set A to include all of what one would consider to be in K. In other words, the requirements can be pushed arbitrarily far into the problem domain, given that enough members of A are added to link them to the specification.

As an example of this second advantage, consider a piloted aircraft where the aircraft need take no trajectory action when it crosses a minimum altitude threshold, but it must ensure that the pilot knows the threshold has been passed. In current practice, the requirements would often state something like:

"The system must raise an alert if the threshold is passed."

However, that is not the true requirement of the system. The *real* requirement is:

"The system will cause the pilot to be aware that the threshold has been crossed."

With the new requirement, the focus has shifted from system design (the alert) to the customer's need (pilot awareness). The *specification* then might become:

"The system must raise an alert if it detects that the threshold is passed."

Distinguishing between the two now exposes the need for attention to be paid to the user interface. The relationship between a system's alert and pilot awareness is not known at this point; it must be stated explicitly as a member of A. In this way, it takes domain assumptions that are often left implicit—and are often incorrect—and forces them to be documented formally.

5 Requirements and Specification in TRIO

One advantage of separating requirements from specification is that two different languages can be used—one that is more suited to application engineers, and one more suited to software engineers. This does not have to be the case, however, and it is simpler to check properties over statements in the same formal language. With this in mind, we used TRIO—Tempo Reale ImplicitO, [CCCMMM99]—as both our formal requirements language and our formal specification language.

At its core, TRIO is a temporal logic with a linear notion of time, and was designed specifically to support documentation of requirements in terms of physical, i.e., real-world, time. It provides facilities for constructing formulas that describe the required/admissible behavior of phenomena, and hence constrain what may happen at particular time instants or over time intervals. In TRIO, the perspective on time is always in terms of the implicit *now*, with other points in time described in terms of their distance from *now* using the *Dist* operator. For example, let us consider the altitude example introduced in Section 2, and suppose that there is in fact a fixed delay *D* between when an altitude value is true in the real world and when such value is actually made available to the application. Such delay could be described by the following TRIO formula:

```
all a(r_alt = a -> Dist(s_alt = a, D));
```

Dist is the only basic temporal operator of the TRIO language; however, a number of derived operators are defined from *Dist*, through the usual first-order logic constructs. For example, the *Alw* operator is used to state that a property holds in every instant (i.e. *always*), while the *WithinF* (*WithinP*) operator is used to state that some property must hold (have held) within a certain future (past) interval (a more detailed list of the TRIO operators is available elsewhere [CCCMMM99]). For example, if the delay between when an altitude value is true in the real world and when this is made available to the application were not *exactly D*, but were simply *bounded* by a constant *D*, one could describe this dynamics through the following TRIO formula:

```
Alw(r_alt = a -> WithinF(s_alt = a, D));[1]
```

One of the major advantages of TRIO is its ability to document real-time requirements in an elegant way. In this work, we use TRIO axioms to document real-time properties, and compose them into ArchiTRIO modules for easier comprehension of the structure of the components and properties stated in the requirements. ArchiTRIO [PRM05] is a UML-oriented extension of TRIO; it uses a subset of UML2 [OMG05] concepts and notations (e.g. structured class, port, interface) to define structural features of systems, and TRIO formulas to describe their dynamics. In fact, as it will be pointed out in Section 7, from a graphical point of view there is very little difference between UML and ArchiTRIO, and all graphical elements that ArchiTRIO retains from UML have the same meaning as in the latter language (albeit in ArchiTRIO such meaning is defined *formally*). Then, the modular structure of ArchiTRIO enables system components to be represented and their interfaces clearly defined; and the temporal model of TRIO then enables a developer writing the specification to set out the requirements' real-time properties very precisely.

---

[1] In a TRIO formula, all free variables are implicitly universally quantified; this holds also for the implicit time, which entails that TRIO formulas are implicitly temporally closed with the *Alw* operator. Hence, the two formulas of this Section could have been written as "`r_alt = a -> Dist(s_alt = a, D))`, and "`r_alt = a -> WithinF(s_alt = a, D)`", respectively.

6 Illustrative Example: Runway Safety Monitor

In order to illustrate our theory of requirements and specification, and to show how formalization of the two can be used in practice to improve developed software, we applied the theory to a "real-life" example: NASA's Runway Safety Monitor (RSM) [Gre00]. We describe the application briefly here, and in the remainder of the paper we discuss the construction of the system's requirements and specification and how the different elements fit the enhanced reference model.

The RSM is part of a larger system, the Runway Incursion Prevention System (RIPS). RIPS is a prototype system designed to address the problem of *runway incursions*, situations where obstacles are present on a runway in such a way that they could interfere with aircraft taking off or landing. The key goal is to assist pilots in maintaining adequate *separation* of their aircraft, in other words, to maintain adequate distance between each aircraft and any obstacles in its flight path. The distance required for separation depends on various factors, including relative size of two aircraft, but these factors are largely abstracted away in our model. The specific rules defining when separation has been violated are set out in the algorithm implemented in the RSM, documented elsewhere [Gre00].

We began work on our example problem by constructing the requirements for the RSM system. We had two major documents available to us: (1) a NASA technical report by David Green, the system's developer, describing the problem of incursion, the algorithm used to detect incursions, and flight test results of the implemented system [Gre00]; and (2) the C source code for the implemented prototype system, provided to us by NASA. We chose to reverse engineer the source code, with guidance from the technical report, to separate the requirements from the specification and the implementation.

7 RSM Requirements

7.1 e_h, e_v, and r_v

The requirements, as documented for the system in the technical report, can be loosely conveyed by the following excerpt from the technical report [Gre00]:

> The Runway Safety Monitor (RSM) was developed by Lockheed Martin in support of NASA's Runway Incursion Prevention System (RIPS) research. ... RSM was developed as a component of the Integrated Display System (IDS), an experimental avionics software system for terminal area and surface operations developed by Lockheed Martin [1][2]. ... The advanced capabilities of IDS and RSM provide pilots with enhanced situational awareness, supplemental guidance cues, a real-time display of traffic information, and warnings of runway incursions in order to reduce the possibility of runway incursions while also improving operational capability. ...
>
> A runway incursion is defined by the Federal Aviation Administration (FAA) as: "any occurrence at an airport involving an aircraft, vehicle, person, or object on the ground, that creates a collision hazard or results in the loss of separation with an aircraft taking off, intending to take off, landing, or intending to land." In other words, a runway incursion occurs when the use of a runway results in a conflict between an aircraft taking off or landing and other traffic, i.e., aircraft, vehicle, person, object, that *may* lead to a collision or loss of separation. An incursion is not an accident; it is a hazardous situation that could cause an accident. Consequently, a runway incursion alert, as defined by RSM, is not necessarily a warning of an impending collision. It is a means of notifying the pilot when a hazardous situation on the runway (i.e., incursion) is detected so that evasive action can be taken to avoid an accident.
>
> RSM does not "prevent" incursions but detects incursions as defined above by the FAA and alerts the pilot via IDS visual displays and aural warnings. ...

We inferred that the key world phenomenon in question was *incursion*. The wording in the excerpt, however, is ambiguous with respect to the exact meaning of the word *incursion* (as is often the case with natural language requirements). The FAA definition quoted in the excerpt implies that incursions are a subset of losses of separation, where the subset is defined by whether the loss of separation involves takeoff or landing of some aircraft. The definition is itself unclear because the notion of loss of separation is intended to clarify what constitutes a collision hazard, and these two seem to be distinct concepts in the definition. However, this discrepancy is not inconsistent with the view that, according to the FAA, a *collision* is a hidden environmental phenomenon (it cannot be measured directly) that should be prevented; a *loss of separation* is a visible environmental phenomenon that should be prevented if possible and dealt with if it

[2] See [BGHJ98].

occurs; and an *incursion* is a type of *loss of separation* (making it a visible environmental phenomenon by definition) that should be prevented or dealt with in a particular way.

Green, the report's author, interprets the definition to mean that an *incursion* is a situation in which a *collision* or *loss of separation* might occur. *Collision* and *loss of separation* seem to have the meanings we have ascribed to them, but *incursion* is different. While our interpretation of the FAA definition differs from Green's, his is based on more extensive analysis of supporting literature (including that from the National Transportation Safety Board), and so is likely a more comprehensive view of what the term means to all the stakeholders involved. Because of this, and because reconciling the source code with our formalized requirements would be extremely difficult if our requirements differed from the requirements on which the code was based, we adopt Green's interpretation.

Given this interpretation, we must now determine whether an *incursion* is a member of e_h or of e_v. In the report, Green effectively defines an incursion in terms of the algorithm used to detect it. In general, algorithms are viewed as part of design (e.g., used in a specification) rather than requirements, and so we had to determine whether the algorithm implemented by the RSM is a part of the requirements or a part of the specification.

Because the algorithm is the formal model of what Green means by *incursion*, it embodies r_v and, thus, is a member of e_v. In practice, there are many instances where requirements are defined formally but are not recognized as such. The classic instance of this is putative theorems, i.e., theorems about properties of a system that are expected to hold if the specification is correct. In developing putative theorems, system designers: (1) construct a specification; (2) determine properties they want to be true over that specification; and (3) prove that the specification satisfies those properties. The theorems represent what the specifier postulates the customer *requires*, and the specification sets out *how the system will achieve it*. The proof is a proof that the specified system will achieve the particular requirement defined by the theorem.

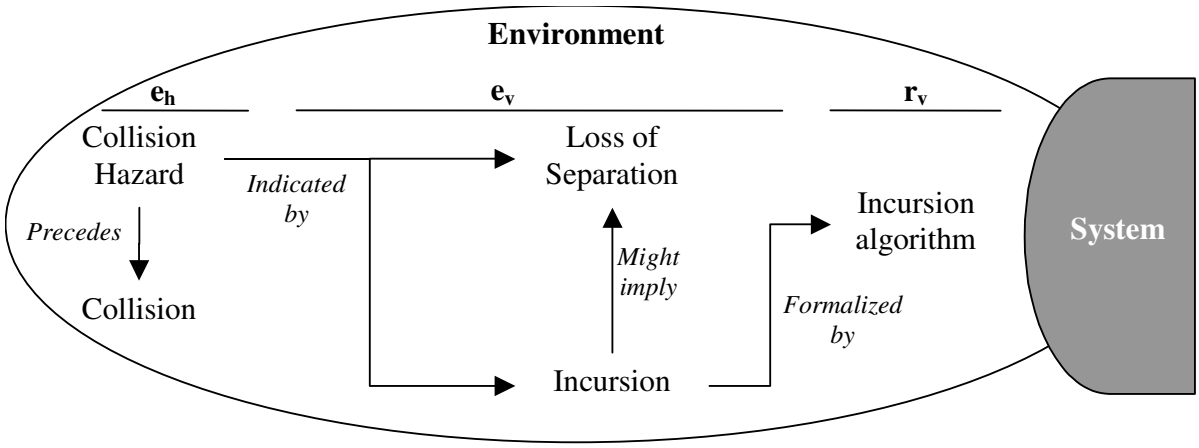

Figure 3 - RSM concepts in the Enhanced Reference Model

It is relatively easy to see how the terms *collision* and *loss of separation* fit into our enhanced reference model. A *collision* is a hidden environmental phenomenon, i.e., a member of e_h, that might be approximately measured by using various specific sensors introduced for the purpose. A *loss of separation* is a visible environmental phenomenon, i.e., a member of e_v, because it is defined within the domain in such a way that any of the various measurement techniques for determining position is adequate to detect it. A *loss of separation* is a necessary precursor to a *collision*. Their relationship to an *incursion*, however, is unclear. To define this relationship, we must introduce a member of e_h from which all of these notions stem: a *collision hazard*. The *collision hazard* is a situation in which, if nothing else in the airspace system changes, a *collision* will occur within an unacceptably short period of time if no evasive maneuvers are taken. Before a *collision hazard* leads to a *collision*, it will lead to a *loss of separation*. A *collision hazard* leads to an *incursion* in particular situations with respect to runways, where *loss of separation* is a poorer predictor but can be used as one element of whether an *incursion* exists. The relationships among these concepts are illustrated in Figure 3.

Our formalized requirements are written solely in terms of e_v since, in this case, we were not able to capture all of the rationale that went into the development of the algorithm. An analysis of the requirements from the domain perspective could formalize the requirement in terms of the hidden variables, introducing additional members of A to show how the specification relates to the more abstract requirements.

The above analysis of e_h and e_v was lengthy, and one might ask whether such analysis is useful. Our analysis (or some less formal version of it) is what anyone validating the system

would need to do (whether they realize it or not) in order to determine whether the system achieves its overall goals. If the relationships among incursion, loss of separation, and collision are unclear, then it is impossible to tell whether the algorithm defining an incursion is sufficient.

7.2 Requirements Components

When extracting our formal requirements model from the existing documents, we were faced with the question of how the components of the source code mapped into the requirements and specification that we were developing. For the requirements, only the aircraft and airport structure were important. There are several other components of the system that we introduce below when discussing the specification, but these components served only to enable the system to detect incursions—not to define what an incursion is and when it must be detected.

We also had to determine the underlying *real-time* requirements for the system. The system has a *requirement* that incursions be detected in a timely manner. In the implemented system, a design decision has been made to execute the algorithm periodically in order to meet this requirement. In practice, this type of requirement is often stated in terms of periodicity and not in terms that are of strict value to the user. The periodicity of the system is a *design decision*, however, and having a user specify it as part of the requirements is not appropriate for two primary reasons. First, it introduces an unnecessary—and potentially error-prone—weaving together of the user's true desires and the way that the system will eventually operate. Second, it overconstrains potential system designs. A more effective place to document such a decision is in the specification: the specification will lay out how requirements will be achieved by the high-level system design which is, in this case, through periodicity.

We expressed the timing requirement for the RSM in terms of the time between when an incursion occurs *in the real world* and the time the RSM onboard component raises an alarm in *the real world*.

7.3 Formal Requirements Model

Having separated the system's requirements, its specification, and its detailed design, we then constructed a formal model of those requirements in ArchiTRIO. The main components of our requirements model are represented through the ArchiTRIO classes shown in Figure 4. The most fundamental one is `Vehicle`, representing the state of each vehicle over time. Class `Vehicle`

represents both aircraft running the RSM program and potential obstacles on the ground. Incursions are detected on each individual vehicle through the RSM component (represented by the ArchiTRIO class with the same name) of that vehicle. As specified by the association between classes `RSM` and `Vehicle`, every vehicle capable of detecting incursions runs an instance of the RSM algorithm.

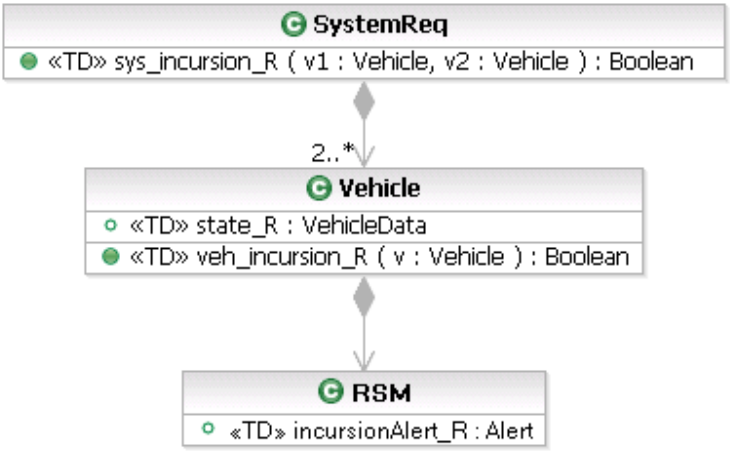

Figure 4 - ArchiTRIO classes representing the core elements of the RSM requirements model.

The elements appearing inside the ArchiTRIO classes (e.g. `veh_incursion_R`) represent the world phenomena associated with the entity modeled by the class; that is, they are variables belonging to the set r_v of Figure 2. The stereotype «`TD`» (where `TD` stands for "time-dependent") that appears next to the name of these elements indicates that they are not to be intended as UML attributes or operations, but as logic predicates and functions (or, in typical TRIO terms, as "items"); these logic items represent the values of phenomena, i.e. variables in e_v, over time, and will then appear in the ArchiTRIO formulas that formalize the requirements of the application. For example, item `state_R` of class `Vehicle` is a variable whose value varies over time (hence the stereotype `TD`), and which represents the *real world* state of the vehicle (position, altitude, speed, etc.); in addition, item `veh_incursion_R` of class `Vehicle` is a predicate that represents when an incursion occurs with the specific vehicle represented by parameter `v`.

The basic requirement for when an incursion is detected can be expressed as the following TRIO axiom of class `Vehicle` (where `ex` is the existential quantifier, and `Lasts` is a temporal operator such that `Lasts(F, d)` defines that `F` holds for (at least) `d` time units starting from the current instant):

```
incursion_detection_R:
  Lasts(veh_incursion_R(v), L_INC) ->
  ex a (WithinF(rsm.incursionAlert_R = a, D_INC));
```

Essentially, this requirement says that whenever there is a "significant" incursion (i.e., an incursion that lasts at least `L_INC` time units, with `L_INC` an application-dependent constant), an incursion alert is raised by the RSM component within `D_INC` time units of when the incursion originally occurs.

The above formula introduces two constants that do not seem to follow from the requirements as we have analyzed them so far. `L_INC` implies that brief incursions (where the definition of "brief" depends on constant `L_INC`) can be safely ignored, and `D_INC` implies that incursion alerts do not have to be raised immediately. These constants must be introduced because the available system components cannot measure environment values perfectly, and so the system is not able to immediately detect or respond to incursions.

In practice, requirements analysts might not realize that these delays are necessary and so might omit such constants. In such cases, the specification will not be able to satisfy the requirements because the system cannot be omniscient. This means that the specifier would have to negotiate with the customer to relax the requirements. Currently, the relaxation process would happen with the specification through retrenchment [BP98]. Precisely identifying the requirements enables that relaxation to be assessed in terms of the problem domain instead of the system design.

Finally, we remark that `veh_incursion_R` and `incursionAlert_R` are members of r_v representing visible phenomena of the environment (members of e_v). They are *not* specification variables. In the next section, we will introduce specification variables to represent whether the machine knows an incursion has happened and whether the machine has ordered an alert to be raised, but in this section we are defining the requirements, i.e., whether an incursion has happened and whether an alert has been raised. While these concepts are very similar, a number of system elements can cause them to diverge. For instance, if the software orders an aural alert to be raised, and the speaker malfunctions, then the logical values of these two formal symbols are not the same. The differences must be carefully analyzed by system designers and domain experts if an argument is to be made about the system's validity.

8 RSM Specification

8.1 Specification Variables

To construct our RSM specification, we created a separate formal model. Because the system does not automatically know its state or the state of other vehicles, the specification contains three significant new system components to acquire this information and transmit it through the system: (1) sensors on each aircraft that determine the aircraft's position; (2) a broadcast mechanism that transmits the aircraft's report of its position to other aircraft and receives the incoming broadcast data from other aircraft; and (3) the `IntegratedDisplaySystem` component that shows alerts to pilots. These components are design decisions at the system level.

Here, we present an excerpt of the specification by focusing on the `Vehicle` class and its components, looking at their different elements. Figure 5 shows the UML2/ArchiTRIO Structure Diagram detailing the components of class `Vehicle`. The machine variables of the specification `Vehicle` class (i.e., its specification variables) are spread through its components. For example, the `RSM` class contains the following machine variables:

- `isIncursion_S`, which represents whether, at a given time instant, the RSM component detects an incursion;
- `target_S`, which is the set of data concerning "target" vehicles (i.e., those other than the one running the RSM) that is used by the RSM to detect incursions;
- `ownship_S`, which is the data concerning the vehicle running the RSM that is collected by the RSM itself (through the vehicle's sensors).

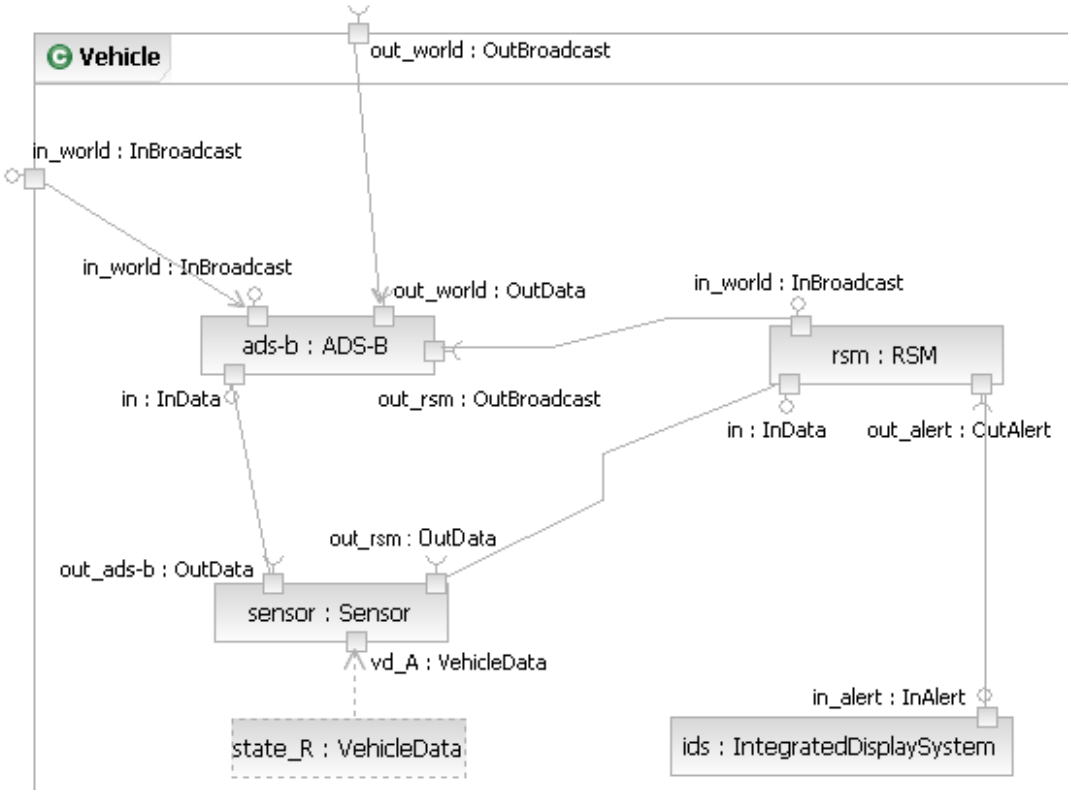

Figure 5 - Structure of class `Vehicle`.

These variables are those that convey the system's knowledge of the requirements phenomena (i.e., the variables of the requirements `Vehicle` class):

- `state_R`, which represents, for each vehicle, its relevant data (i.e. altitude, position, etc.);
- `veh_incursion_R`, which represents whether, at a given time instant, there is an incursion (independent of whether this is detected).

Note that some of these variables are closely related to the variables of the `Vehicle` class in the requirements. The similarity in naming reflects the linked purposes of these variables: variables ending in `_R` are members of r_v, and variables ending in `_S` are members of s_v that model the same members of r_v. Some variables in s_v do not map—either directly or as a group—to any requirements variables; for instance, the operations offered and required by the various

components through their ports[3] (e.g., operation `broadcastData`, which is exported by class `RSM` through interface `DataBroadcast` of its `in_world` port). These represent additional details in the system description that are introduced in the specification to make it both sufficiently detailed to meet the requirements and a realistic, abstract description of the system behavior.

8.2 Uncertainty Introduced in the Specification

As we explained above, the definition we use for specification includes the high-level structure of system components that enables the system to meet its requirements. The RSM's requirements state the conditions under which vehicles are considered to *be* in an incursion situation. The specification states the conditions under which vehicles *will detect* the incursion. This is why we had to relax the original requirement: there is some uncertainty introduced because of the error in each vehicle's knowledge about the state of the system. Because of our separation between requirements and specification, we enable customers to define when a system is in an incursion, and the bound on error within which the system must detect the incursion. The specifier, then, will architect the system to meet the requirement.

In the RSM system architecture, the vehicles share information about each other through a broadcast mechanism. Each vehicle periodically polls its sensors to detect its state and periodically sends its state to the broadcast mechanism, which in turn periodically broadcasts the state of all vehicles in the system to all of those vehicles. Thus, error due to staleness of data is introduced basically in three places:

1. the delay between when the state was true of the vehicle and when the vehicle is able to use the data (the data has been transmitted by the sensors);
2. the delay between when the data is available for use by the vehicle and when the vehicle sends it to the broadcast mechanism; and
3. the delay between when the broadcast mechanism receives the data and when the other vehicles receive the periodic broadcast.

Also, as explained earlier with our altitude example, any system (such as the RSM) that makes control decisions based on data coming from sensors is susceptible to measurement

[3] We note that the concepts of "port", "interface" and "operation" in ArchiTRIO are the same as in UML2, the only difference being that in ArchiTRIO they have been given a formal semantics [PRM05].

errors; that is, its behavior is (possibly significantly) influenced by the differences that exist between the value read by the sensor and the actual value of the quantity when the measurement is taken. While in this paper we do not deal with measurement errors, they could be easily included in a further, more refined specification of the RSM system.

We take into account timing and measurement errors when linking the sets r_v and s_v through the set A. A is stated as a further set of TRIO formulas that document the relationships between related variables in the two sets. These formulas state the potential error in the variables of s_v as compared to the phenomena whose variables are modeled in r_v. We describe the way that we link A to the specification in more detail below.

8.3 Form of the Specification

The RSM specification is a separate entity from the requirements. It has a similar form to the requirements, and even a number of components that map directly to requirements components, such as the `Vehicle` and `RSM` classes that appear in both the requirements class diagram of Figure 4 and among the specification classes. When they occur, direct mappings help the verification stage, but they are a coincidence and are not necessary. The broadcast mechanism, for example, is a component that exists only in the RSM specification.

We chose to distribute the formulas of A across the different classes in the specification to take advantage of ArchiTRIO's support for separation of concerns. To be able to state the relationships, variables from both the requirements and the specification had to be included in the formulas of A associated with the specification classes. For example, the following formula (which belongs to class `RSM`) links the real-world event that the RSM's state variable `incursionAlert_R` takes on a defined value, `a`, with the system event that the RSM component invokes an operation `raiseAlert` (of which `out_rA` is an instance) of port `out_alert`:

```
incursion_alert_def_A:
  incursionAlert_R = a <-> ex out_rA (out_rA.recv(a));
```

As another example, consider the following. Each aircraft's sensors will estimate the real-world state values of that aircraft and then send these estimates to the RSM algorithm and to the broadcast system within `D_DS` seconds of their being true of the aircraft. This axiom is formalized in the ArchiTRIO formula below, where `rsm_rD` is an instance of operation `recData` of `Sensor`'s

port `out_rsm`; `rsm_rD.invoke` is the operation's invocation event; `vd` is the actual state of the vehicle (which corresponds to the value represented by item `state_R` of class `Vehicle` shown in Figure 4); and `d` denotes the vehicle's state:

```
data_sent_def_A:
  rsm_rD.invoke(d) -> WithinP(vd_A = d, D_DS);
```

This axiom is included in the ArchiTRIO class for the `Sensor` specification component. Similarly, other delay axioms are introduced for other time components at the appropriate places in the specification, completing the set A for the RSM and distributing it where it can be easily validated against its corresponding element in the system design (such as the use of a particular sensor).

Notice that the specification of a vehicle includes a `Sensor` component, which does not appear in the system requirements; this is possible since, as mentioned above, the components of the specification need not be in a direct relationship with those of the requirements. However, as formalized by axiom `data_sent_def_A`, a sensor needs to "read" the real-world state of the vehicle in order to then send appropriate data to the RSM; to formally state this property, variable `state_R` of `Vehicle` needs to be made "available" in the appropriate class (in this case, class `Sensor`). This is formally achieved through variable `vd_A` of class `Sensor`, which is equivalent to variable `state_R` of its containing class `Vehicle` (as shown in Figure 5, where the two elements are linked through a connector). As the suffix indicates, `vd_A` is in fact an element of set A, since it formally bridges specification and requirements. One can refer to such elements when there is no direct mapping from the elements of the requirements to those of the specification, but a correspondence between the variables is necessary.

The distinction between the two different types of formulas—those that are members of S and those that are members of A—is straightforward: those that include any requirements variable are members of A, and those that include only specification variables are members of S. An implementer wanting to study only S, without any reference to A, could use a filter to remove these formulas.

Any omitted delays can be detected through review of these axioms: if the sensor specification did not include a delay, then the axiom would say that the state is transmitted instantaneously, a situation that a sensor designer could identify as a false assumption. If the

specification alone were written, there would be no formal need to create the axiom, and identifying the *implicit* assumption that sensor state was updated and broadcast instantaneously would be much more difficult. If the mistake were then not detected through an informal validation process, it: (1) might be caught when the source code was written and corrected in the program but not the specification, leaving the specification out of date and possibly failing to identify all of the ramifications of the change; (2) might be caught during testing and changed at great expense; or (3) might not be caught, possibly leading to a failure of the system.

Finally, while specifications are written using members of s_v, if the system is not yet implemented, the exact values for those variables may not be known. For instance, the broadcast mechanism will introduce some delay, but the specific delay it introduces depends on which hardware is used. Specifying exact numbers at this stage can overconstrain potential system designs, and so introducing variables with necessary constraints (such as a maximum value for a delay) but without specific values can be useful. Our RSM specification includes a number of these variables that represent the delays described above. These constraints are also members of A since they relate specification variables to requirements variables, but they are not *axiomatic* constraints because they must be *made* true by the system. Instead, we document them as TRIO *assumptions*, whose semantics require that the assumptions be shown to be true of a lower-level design [FRMM06]. These formulas can be assumed in validation activities undertaken before design is complete, even though their exact values might not be known.

9 Conclusion

Current use of the terms *requirements* and *specification* does not clearly distinguish between the two. Because of this, validation—one of the most difficult problems in software engineering—is a harder problem than need be. Separating the two documents and structuring their roles and relationship has the potential to profoundly improve software development practice.

This paper has made two contributions. The first is to refine the reference model of Gunter et al. into one that cleanly separates concepts as they exist in the world and the representation of those concepts in the system. We then added a new set, A, into the reference model to enable domain experts to document the relationship between each concept and its representation.

The second contribution is to show how the concept of validation can be decomposed, and show some of the ways that it can be accomplished. We described three potential strategies for

validation. In a companion paper [FRSMK] we discuss the complementary issue of further exploiting formal methods by raising the level of formalization up to the requirements level and by formally verifying the correctness of the specification against the requirements.

Acknowledgments

The authors would like to thank Michael Jackson and Bashar Nuseibeh for their helpful comments and suggestions on an earlier draft of this paper. This work was partially supported by the Short Term Mobility program of the Italian CNR (Consiglio Nazionale delle Ricerche), in part by NSF grant CCR-0205447, and in part by NASA grant NAG1-02103.

References

[BP98] R. Banach and M. Poppleton: "Retrenchment: an engineering variation on refinement". In *Proceedings of the 2nd International B Conference*, Lecture Notes in Computer Science, 1393:129–147, 1998.

[BGHJ98] S. O. Beskenis, D. F. Green, P. V. Hyer, and E. J. Johnson: "Integrated Display System for Low Visibility Landing and Surface Operations". Technical report NASA/CR-1998-208446, 1998.

[CCCMMM99] E. Ciapessoni, A. Coen-Porisini, E. Crivelli, D. Mandrioli, P. Mirandola, and A. Morzenti: "From formal models to formally-based methods: an industrial experience". *ACM Transactions On Software Engineering and Methodologies*, 8(1): 79-113, 1999.

[CHR91] Z. Chaochen, C. A. R. Hoare, and Anders P. Ravn. "A calculus of duration", *Information Processing Letters*, 40(5):269–276, 1991.

[FMMPRS04]. C. A. Furia, D. Mandrioli, A. Morzenti, M. Pradella, M. Rossi, and P. San Pietro: "Higher-Order TRIO". Technical Report 2004.28, Dipartimento di Elettronica ed Informazione, Politecnico di Milano, 2004.

[FRSMK] C. A. Furia, M. Rossi, E. Strunk, D. Mandrioli, and J. C. Knight: "Bringing Formal Methods Up To the Requirements Level", in preparation.

[FRMM06] C. A. Furia, M. Rossi, D. Mandrioli, and A. Morzenti: "Automated compositional proofs for real-time systems". *Theoretical Computer Science*, 2006. (To appear).

[Gre00] D. F. Green: "Runway Safety Monitor Algorithm for Runway Incursion Detection and Alerting". Technical Report, NASA Langley CR 211416, 2002.

[GGJZ00] C. A. Gunter, E. L. Gunter, M. A. Jackson, and P. Zave: "A Reference Model for Requirements and Specifications". *IEEE Software* 17(3):37-43, 2000.

[HJLNR02] J. G. Hall, M. Jackson, R. C. Laney, B. Nuseibeh, and L. Rapanotti: "Relating software requirements and architectures using problem frames". In *Proceedings of the 10th IEEE Joint International Conference on Requirements Engineering*, pp. 137-144. IEEE Computer Society, 2002.

[HR03] J. G. Hall and L. Rapanotti: "A Reference Model for Requirements Engineering". *Proceedings of the 11th IEEE International Requirements Engineering Conference (RE'03)*, 2003.

[HRJ05] J. G. Hall, L. Rapanotti, and M. Jackson: "Problem frame semantics for software development". *Journal of Software and Systems Modeling*, 4(2):189-198, 2005.

[IEEE98] IEEE Recommended Practice for Software Requirements Specifications. IEEE STD-830, 1998.

[Jac00] M. Jackson: *Problem Frames: Analyzing and Structuring Software Development Problems*. Addison-Wesley, 2000.

[JHJ06] C. B. Jones, I. Hayes, and M. Jackson: "Specifying Systems that Connect to the Physical World". Technical Report, CS-TR-964, University of Newcastle upon Tyne, 2006.

[MT01] S. P. Miller and A. C. Tribble: "Extending the Four-Variable Model to Bridge the System-Software Gap". In *Proceedings of the 20th Digital Avionics Systems Conference (DASC'01)*.

[OMG05] Object Management Group: "UML 2.0 Superstructure Specification". Technical Report, OMG, formal/05-07-04 (2005).

[ORS92] S. Owre, J. M. Rushby, and N. Shankar: "PVS: A Prototype Verification System". In *Proceedings of CADE-11*, Lecture Notes in Computer Science, 607: 748–752, 1992.

[OS99] S. Owre, and N. Shankar: "The Formal Semantics of PVS". Technical Report CSL-97-2R. SRI International. March 1999.

[PM95] D. L. Parnas and J. Madey: "Functional Documents for Computer Systems". *Science of Computer Programming* 25(1):41-61, 1995

[PRM05] M. Pradella, M. Rossi, and D. Mandrioli: "ArchiTRIO: a UML-compatible language for architectural description and its formal semantics". In *Proceedings of FORTE'05*, Lecture Notes in Computer Science 3731: 381-395, 2005.

[Rus89] J. Rushby: "Kernels for Safety?". In T. Anderson, editor, *Safe and Secure Computing Systems*, chapter 13, pages 210–220. Blackwell Scientific Publications, 1989.

[ZJ97] P. Zave, and M. A. Jackson: "Four dark corners of requirements engineering". *ACM Transactions on Software Engineering and Methodology* 6(1):1-30 , 1997.