Raising Formal Methods To The Requirements Level

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

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

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

Abstract

In contrast with a formal notion of specification, requirements are often considered an informal entity. In a companion paper [SFRKM06], we have proposed a reference model that provides a clear distinction between requirements and specification, a distinction not based on the degree of formality. Using that notion of requirements, this paper shows how *requirements* as well as specifications can be *formalized*.

The formalization of requirements allows one to "lift" the well-known practices of formal analysis and verification from the specification/implementation level up to the highest level of abstraction in the development of a software product. In particular, we show how a formal validity argument can be constructed, proving that the formal specification satisfies its formal requirements. These ideas are demonstrated in an illustrative example based on a runway incursion prevention system, which was also partially presented in [SFRKM06]. Although our results and methods are of general applicability, we focus especially on the real-time aspects of the example; in order to support real-time formalization and reasoning, we exploit the ArchiTRIO formal language in a UML-like environment.

1 Introduction

In the realm of computing and software engineering, traditional formal methods, such as those embodied in VDM [Jon90], Z [Spi92] and B [Abr96], focus on the specification and verification of implementations. Though highly desirable, an implementation is one of the last artifacts that is produced in the whole process of system development. By contrast, this paper is about formalizing requirements, the first artifact that is usually produced. We present a comprehensive approach to formalizing requirements, and discuss the various advantages that result from this formalization.

Certainly, the need to raise the abstraction level of formal documentation and analysis towards the higher phases of the process has been widely acknowledged, and this goal has been pursued both by adopting—and adapting—existing formalisms and by introducing new ones. A distinguishing feature of a formalism suitable for dealing with top-level requirements is the possibility of describing different entities that are quite heterogeneous in nature, such as a sequence of human actions and the speed of a vehicle. This is not the case with traditional mathematical machineries such as differential equations or finite state machines.

Furthermore, as pointed out in our earlier work [SFRKM06], when moving from requirements to design specification, one must distinguish clearly between physical quantities and measured quantities. A particularly relevant special case of such a difference is provided by the "time" quantity: for instance, in most real-time systems requirements are expressed in terms of the physical time—often modeled as a real value—whereas system design is based on time as measured by one or more clocks, providing a discrete value. Most formalisms, however, assume a rigid formalization of time either as a continuous variable as in differential equations or as a discrete one as in normal abstract machines adopted in computer science and software

engineering.[1]

The formalization of requirements that we present is independent of any formalization of the specification. Formalizing requirements implies that a domain expert can set out what the system must do in a way that is more natural to the expert and is not tailored to the needs of software development.

In order to demonstrate raising the abstraction level of formalization, we present an illustrative example. The example is based on the *Runway Safety Monitor* (RSM) [Gre00], part of a prototype system to prevent airport runway incursions. Through the development of the requirements and specification of part of that system, the example illustrates how the approach might be applied in practice. Although the range of applicability of the method is broad, we focused our analysis on the real-time aspects of the application.

We constructed both the requirements and specification for the RSM, and we developed a proof that the latter implied the former using the ArchiTRIO formal language [PRM05] in a UML-like environment. We describe the various artifacts produced and our experience producing them, including defect detection in our specification. This example was also used in a companion paper [SFRKM06] that had different objectives and content; the present paper builds on some of the results of the other. In the remainder of this paper we refer to [SFRKM06] whenever possible, but we also repeat some of the material in order to make the present paper self-contained.

The remainder of this paper is organized as follows. Section 2 exemplifies the role of formalization of requirements within the development process of a system. Section 3 presents the

[1] See Furia et al [FMMR] for a comprehensive analysis of the literature on time formalization in computing machinery.

overall view of requirements and specification, explaining the role of formalization. Section 4 presents the TRIO formal language, and its ArchiTRIO extension. Section 5 introduces the Runway Safety Monitor (RSM) example, through which our approach to formalization is demonstrated. Then, Section 6 illustrates the formalization of the RSM requirements, and Section 7 the formalization of its specification. Section 8 shows how the verification of the specification against the requirements has been formally carried out. In particular, Section 9 highlights a disagreement between requirements and specification found in an earlier version of the formalization, and how it has been overcome. Section 10 presents works related to the present one, and Section 11 concludes.

2 On Formalizing Requirements

It is fairly common in engineering for the first decisions about a new product—indeed, its requirements—not to be formalized; formalization only occurs at later phases of the design process. This is probably due to the fact that early decisions are based on a mixture of technical and non-technical aspects, including economical and social factors. For instance, when a road intersects a railway the first decision is whether to build a bridge for one of them or to let them cross; in the latter case one should then decide how the crossing is to be managed: whether through a gate that prevents cars from crossing the railroad when a train is coming, or simply through a traffic light, leaving to the car drivers the responsibility to stop at red light. Such fundamental decisions depend on heterogeneous factors such as traffic intensity, drivers' behavior, construction costs, etc. which are more difficult to be formally described, evaluated, and compared than, say, the construction of a bridge, or of an automatic gate. Nevertheless a rigorous analysis of these aspects is as important as a technically sound design and implementation of the chosen "device" to be built.

As a consequence, the first formalized document about a product to be developed is often a mixture of *requirements* and *design specifications* where several early analyses and decisions are blurred or left implicit. Often, it is merely the ability to formalize a particular piece of information that is used to classify that information as a requirement or a design specification. For instance, the above example of the intersection between road and railway has been widely used in the literature as a benchmark to assess the properties of formal methods for real-time computing [HM96]. In most cases, however, the proposed formalizations describe the needed properties of the gate used to manage the railroad crossing, without noticing that the real requirement for the intersection is to avoid (or to minimize the likelihood of) accidents between cars and trains at "reasonable costs". Formalizing such a requirement however, is much more difficult than stating at which time exactly the gate must be closed to guarantee that no car can cross the railroad when a train is arriving.

In a companion paper [SFRKM06], we introduced an enhanced reference model in which the distinction between requirements and design specification is clearly and unambiguously stated, independently of the notion of formality. The use of formal specifications is not common, but it is at least widely documented in the existing literature. In this paper, we advocate that the level of formalization should be further raised beyond design specifications and should cover at least part of the requirements. Inevitably, a part of what we call "requirements" will remain informal, albeit stated as precisely as possible, because interpretation of variables has to be in natural language. What we show in this paper is that some requirements can be formalized, and that the formalization brings several advantages in terms of what one can assess at a still very abstract level.

First of all, the formalization of requirements supports validation of the system because it

enables the domain expert who writes the requirements to validate the formal model purely in terms of the concept for the operation of the system. If requirements are not formalized, then the expert must either validate the model by inspection, or wait until it has been made more complex by being combined with high-level system design in the specification. Formalizing requirements enables him or her to validate the formal model in a way that is more natural and is not tailored to the specific needs of software development.

Formalizing requirements also permits greater precision in the statement of the requirements, a property that is generally useful but is especially important in documenting real-time requirements. Real-time requirements are often created with design details in mind; a common example is synchrony vs. asynchrony. A system is not *required* to be synchronous or asynchronous, although in many cases the *actual* requirements might be implementable with only one of the two design choices. Validation is much easier when it does not have to consider the intricacies of a particular development strategy. Development strategies can be complex, however, so it is often important to include them in a *specification*.

With both requirements and specification formalized, a formal argument can be constructed which proves that the formal specification satisfies its formal requirements. This formal argument does not prove that the system is valid: if the requirements are invalid, then the specification could be no better. Nevertheless, providing this formal link between requirements and specification identifies the specific role of validation: it is the process of assuring that the requirements, not the specification, document precisely what the customer wants the system to do.

The formal argument also offers the opportunity to eliminate, through analysis, a wide class of potential errors that might either be present in the requirements or arise in the development of

the specification. In summary, formalizing the requirements and the specification makes it possible to "lift" the well-known practices of formal analysis and verification, practices that typically pertain to the lower abstraction levels of specification and implementation, up to the highest level of abstraction in a system development and documentation.

3 Formal and Informal Requirements, and Specification

This section discusses the "big picture" of requirements and specification; Figure 1 serves as a reference for the concepts and ideas.

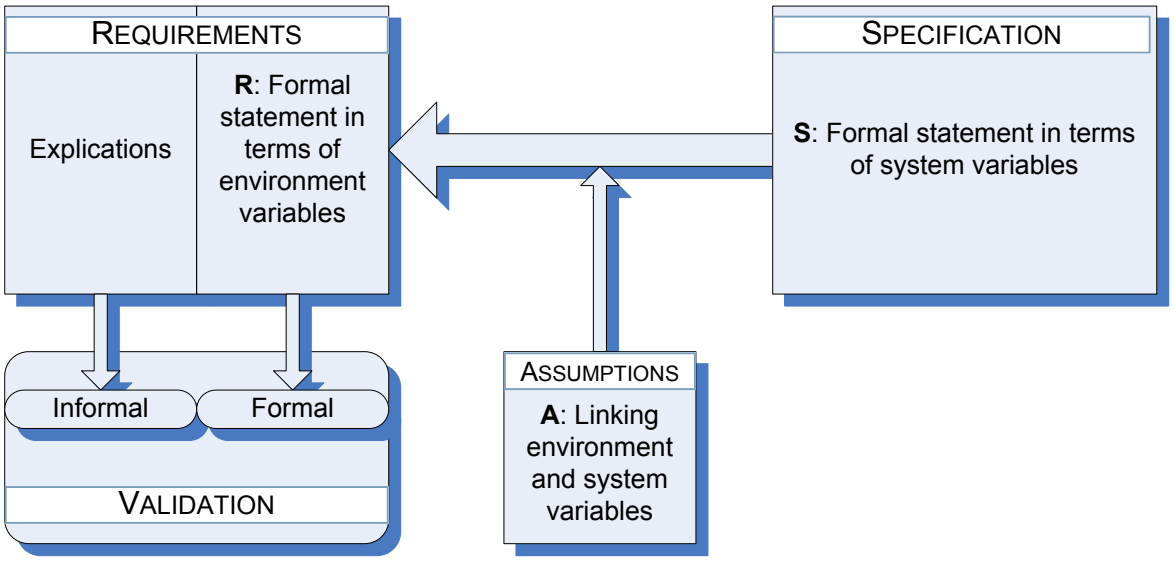

Figure 1 - Overview of the relationship between requirements and specification

3.1 Requirements and Validation

In our companion work [SFRKM06], we introduced an enhanced reference model that is based on the work of Gunter et al. [GGJZ00]. In this paper, the term "reference model" refers to our enhanced model. In our earlier paper, we marked a clear distinction between requirements and specification. In particular, we showed how requirements are the statement of the expected functionality of a system stated solely in terms of *world variables*, i.e., variables found only in the real world and referred to as the set e_v in the reference model.

The *requirements* of a complex system are a statement of exactly what the system has to achieve in terms of world variables. Generally speaking, this involves a precise explanation of what each variable represents in the real world, and in what sense that is relevant to the system being described. Such explanations can only be informal because they have to document links between world phenomena and variables representing them, and world phenomena can only be described in natural language. Once the links have been documented, however, much clarity can be gained by using the variables when defining system functionality.

To illustrate these ideas and to summarize the aspects that are relevant to the remainder of this paper of the reference model we introduced earlier [SFRKM06], we present a simple example. Our example involves the requirements of a car speed-limit monitor; obviously, this includes a notion of speed. A car's speed is a world phenomenon, which would be embodied by some environment variable *r_speed*. Thus, a part of the requirements would document this choice, stating explicitly what *r_speed* represents (i.e., the speed of the car), and how (e.g., as measured with respect to the ground, in miles per hour).

Such informal, defining statements would then make it possible to express *formally* the expected *relations* among these world variables that the system under development is supposed to maintain. For instance, still with reference to the same example, suppose that the monitor should issue an alarm whenever the speed goes above a given threshold r_{max}. This could be expressed formally, once we have specified what *r_speed* is. Assuming that there exists another variable *r_beep* representing the phenomenon of an aural alert being raised in the world, the aforementioned requirement could be expressed formally with the logic formula:

$$r\_speed > r_{max} \rightarrow r\_beep$$

We note that this example is *partial*, in that, being a sufficient condition, it does not impose

any restriction on issuing alarms when the speed is below the maximum threshold. This would be dealt with in practice by other requirements, such as *utility* requirements.

Continuing in this manner throughout the requirements documentation process, we get to a set of requirements that are stated formally, and which are complemented by informal explanations, clarifications, and links to the real world. A *validation* activity could then take place on this document, ensuring that the formal and informal parts match in the right way, and thus that the original intended meaning is preserved by the formalization. Thus, the goal of the validation process is much more clearly stated when (a part of) the requirements are formalized.

3.2 Formal Specification and Assumptions

We argue that the specification of a system is an abstract representation of it stated entirely in terms of *machine variables*, i.e., variables that can take part in computations within the machine and referred to in the reference model as the set s_v [SFRKM06]. In particular, some machine variables are the measured counterparts of the world variables appearing in the requirements.

Returning to our speed monitor example, the specification could be expressed in terms of the machine variables *s_speed*, s_{max}, and *s_beep*, corresponding to the speed as measured by the car's speedometer, the program threshold for issuing an alarm, and a system representation of a beep. Thus, the specification could be stated simply as:

$$s\_speed > s_{max} \rightarrow s\_beep$$

Then, one must provide an explicit link between the world variables and their system counterparts. This link can, and should, also be represented formally, so that the difference between world variables and machine variables, such as the effects of the measurement inaccuracy or sensing errors, can be clearly understood. Most importantly, this formal link permits the consequences of these differences on the validity of the specification to be assessed

unambiguously and quantitatively.

Returning to our example, suppose that the measured speed *s_speed* always has a value within 0.5 miles per hour of the "real" speed *r_speed* and that the system alarm always represents the occurrence of an alarm in the world. This can be formalized as:

$$r\_speed - 0.5 <= s\_speed <= r\_speed + 0.5$$

$$s\_beep = r\_beep$$

With these relations in mind, the impact of different choices for the value of the specification threshold s_{max} are immediately clear. In fact, if one sets $s_{max} = r_{max}$, then the requirements cannot be satisfied by any implementation because if *s_speed* is lower than the actual speed *r_speed*, it will fail to trigger an alarm when the latter is above the threshold while *s_speed* is not. However, if one sets:

$$s_{max} = r_{max} - 0.5$$

then, whatever the measurement error is (provided it is within the stated bound of ± 0.5 mph), the requirements will be satisfied by the specification.

3.3 Implication Proof

After we have formalized the specification, the relevant part of the requirements, and the relations between the world variables and the associated machine variables, we can construct a formal argument that the specification implies the requirements. The particular techniques for achieving this step depend on the kind of formalisms that have been adopted for the various parts. If the same logic formalism is used for all artifacts—as we did in speed-monitor example—then the formal argument is a proof in the classical logic sense.

The speed-monitor example was specified using simple arithmetic predicates. The proof that the specification implies the requirements is as follows. If $r\_speed > r_{max}$ then $r\_speed - 0.5 >$

$r_{max} - 0.5$ and *s_speed* >= *r_speed* $- 0.5 > r_{max} - 0.5 = s_{max}$ implying that *s_beep* is true as required. From the equivalence between *s_beep* and *r_beep*, we conclude that *r_beep* is true whenever the speed is above the prescribed maximum threshold.

Naturally, in practice, other verification techniques could be used depending on the chosen formalisms.

4 Requirements and Specification in TRIO

The choice of the formalisms to be used depends, in general, on the aspects of the application that one would like to formalize. Different formalisms can be used for the requirements and the specification but, for our illustrative example, we chose to use the same formal language, one that is suitable for formalizing both the requirements and the specification, and that is sufficiently expressive to deal with real-time constraints in a natural way. We chose TRIO (Tempo Reale ImplicitO, [CCCMMM99]), with its ArchiTRIO [PRM05] extension. This section describes the language basics, and introduces some of the details that arise when dealing with both the requirements and the specification of a real-time system.

4.1 TRIO and ArchiTRIO Basics

TRIO is a temporal logic with a linear notion of time that was designed specifically to support documentation of requirements in terms of physical, i.e., real-world, time. Nonetheless, its generality and flexibility allow the efficient expression of statements involving machine time, such as in a specification (see also Section 4.2).

TRIO 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

instance, in the speed-monitor example suppose that there is a fixed delay *D* between when a speed value is true in the real world and when that value is actually made available to the application. Such a delay could be described by the following TRIO formula:

```
all s (r_speed = s -> Dist(s_speed = s, 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* (resp. *WithinP*) operator is used to state that some property will hold (resp. have held) within a certain future (resp. past) interval (a comprehensive list of the TRIO operators is available in [CCCMMM99]). For example, if the delay between when a speed is true in the real world and when this is made available to the application were not *exactly D*, but were simply *bounded above* by a constant D, one could describe this dynamic using the following TRIO formula:

```
Alw(r_speed = s -> WithinF(s_speed = s, D));
```[2]

TRIO formulas can be used as axioms to document environment and machine real-time properties; TRIO formulas can be composed into ArchiTRIO modules to represent the structure of the components together with their properties stated in the requirements. ArchiTRIO [PRM05] is a UML-oriented extension of TRIO; it uses a subset of UML2 [OMG05] concepts and

[2] In a TRIO formula, all free variables are implicitly universally quantified; this holds also for implicit time, and so 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.

notations (e.g., structured class, port, and interface) to define structural features of systems, and TRIO formulas to describe their dynamics. From a graphical point of view there is very little difference between UML and ArchiTRIO, and all graphical elements that ArchiTRIO keeps from UML retain their semantics, albeit in ArchiTRIO semantics are defined *formally*. 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 precisely.

Once we have formalized a specification and, to the extent possible, the requirements in TRIO (and ArchiTRIO), we can exploit a variety of mechanisms to verify that the specification implies the requirements. In our illustrative example discussed below, we chose to do that by translating the TRIO formalism into the PVS logic language [ORS92], and then using the PVS theorem-proving system to assist and check the correctness proof. This approach to verification with TRIO has been discussed elsewhere [GM01].

4.2 World and Machine Time in TRIO

While broadly applicable to many system properties, the focus of our verification is the *timing* of critical systems. Timing requirements are prominent in a wide variety of systems [Liu00,SLMR05,HS06]. TRIO is appropriate to our study because it is well suited to the expression of timing properties.

We begin our discussion of time by noting again that in TRIO time is *implicit*, that is there is no explicit variable that represents the absolute value at which the truth of a formula is evaluated. This is consistent with what is commonly done in modal logics, where special operators are dedicated to the expression of relations over particular domains, time being a specific instance of these. Thus, time is a variable of our formal system description, even though it serves as an

external reference rather than being presented explicitly.

In considering requirements and specifications, there are (at least) two distinct notions of time:

- *world time*: this is the physical time of the real world, the fundamental reference in describing the dynamics of a system;
- *machine time*: this is an "implemented" time, as seen by a machine which has access to readings of some (in general, imperfect) clock.

Thus, we have two underlying *models* of time, one for world time and the other for machine time. TRIO can deal with both: all that is necessary is to specify, for any formula, in which of the two models the formula is to be interpreted. According to our reference model, requirements should be expressed solely in terms of world time and specifications should refer solely to machine time, and we effect this distinction with the two interpretations in TRIO.

It is useful to adopt different domains to represent different time models in TRIO. For example, whereas world time is typically represented by *real* numbers, machine time is usually discrete and is typically represented by *integer* numbers. TRIO is model parametric, so that its formulas can be interpreted over different time numeric domains. Nonetheless, depending on the degree of abstraction one is dealing with in the specification, one may simply represent the two times in a homogenous manner. In our illustrative example we will pursue this latter option by choosing the real numbers as domain for both world and machine time. Using real numbers for machine time means that the specification effectively has a continuous clock. An implementation, then, will have to contain a clock whose precision and accuracy enable it to satisfy the specification, but both specification and implementation are actually "sampling" time over the machine domain.

Finally, in order to be able to infer timing facts referring to the real world from specification statements referring to machine time, one has to provide explicitly a relationship between world time and the implemented machine time. Not unlike what is done with other variables, the link between world and system instances has to be provided as a set of relations in the set A (see Figure 1 and [SFRKM06]). Note that such relationship basically embeds information about the physical execution times of some processes in the given system. Therefore, it is ultimately an assumption about the performances of the implemented system; this is consistent with our view, also stated in [SFRKM06], that the specification is a form of abstract design.

Again, the stated relationship between the two times can be as sophisticated as it is needed and appropriate. In the example of this paper, we chose a simple "lightweight" solution over more complicated ones since the focus of this work is the formalization of *requirements* and what can be done with that. A detailed and thorough analysis of possible ways of implementing time and their relations with world time is an interesting topic on its own, orthogonal to the scope of this paper. Some aspects of it are the object of other work [FR06,FMMR], as well as of future investigations.

5 Illustrative Example: Runway Safety Monitor

In order to demonstrate how formalizing requirements—as well as specification—is useful in practice to improve developed software, we illustrate the formalization of an example: NASA's Runway Safety Monitor (RSM) [Gre00]. We first describe the application briefly and informally here; in the remainder of the paper, we elaborate the difficulties we faced in constructing the system's requirements and specification, and we describe the solutions we used to overcome those difficulties, and the gained confidence in our understanding of the system.

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 an incursion occurs are set out in the algorithm implemented in the RSM, documented elsewhere [Gre00].

We began work on our 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.

6 RSM Requirements

The requirements elicitation process for the RSM system has been carried out in detail in [SFRKM06]. Here, we just recall its results and proceed to the formalization stage.

The system has a *requirement* that incursions be detected in a timely manner. 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*. The details of the requirement will become clear as we explain our formalization thereof.

6.1 Formal Requirements Model

The main components of our requirements model are represented through the ArchiTRIO classes shown in Figure 2. The 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.

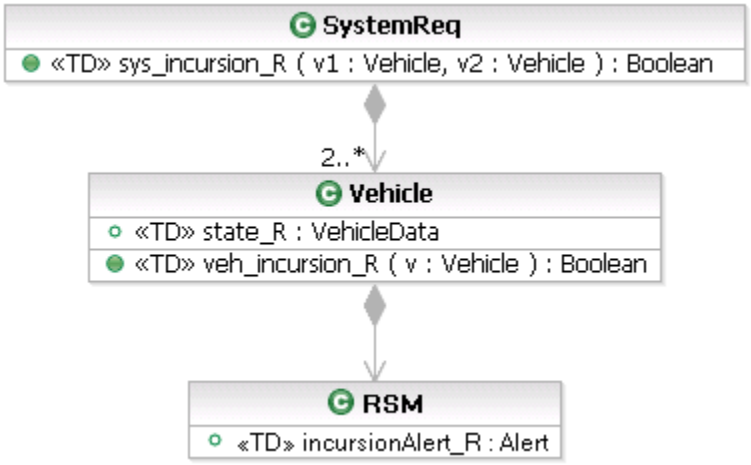

Figure 2 - ArchiTRIO classes representing the core elements of the RSM requirements model.

The elements appearing inside the ArchiTRIO classes (e.g., `veh_incursion_R`) represent the phenomena associated with the entity modeled by the class; that is, they are world variables. 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) 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 vehicle represented by parameter `v`.

Each of the classes of Figure 2 has TRIO formulas defining its real-time behavior. The basic requirement for when an incursion is detected can be expressed as the following TRIO formula 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 *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.[3]

As we have discussed at length elsewhere [SFRKM06], the two constants `L_INC` and `D_INC` introduce a "timing tolerance" in the requirements about the detection of incursion. Such a tolerance is required because the system components cannot perfectly measure environment values.

Finally, we note that `veh_incursion_R` and `incursionAlert_R` represent visible phenomena of the *environment*, not specification variables.

[3] For brevity, we choose not to deal with requirements about not raising an alert if there are no incursion, that is with utility requirements about the absence of false positives. Such requirements could be dealt with separately, using the same techniques we documented here.

7 RSM Specification

The RSM specification contains three system components other than those that are also in the requirements model: (1) sensors on each aircraft, which detect the aircraft's position; (2) a broadcast mechanism, which 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, which shows alerts to pilots. The functionality and timing of each of these components is formalized in the specification through TRIO formulas, some of which we illustrate in the following sections. We also provided formulas linking the specification variables to the requirements variables: these are part of set A in Figure 1 and in the reference model of [SFRKM06]. We note that all specification formulas refer to *machine time*.

In the RSM system architecture, each vehicle regularly polls its sensors to detect its state and sends its state to the broadcast mechanism, which in turn broadcasts the state of all vehicles in the system to all of those vehicles. Thus, error due to staleness of data is introduced 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 illustrated earlier with our speed example, any system that makes control decisions based on data coming from sensors (such as the RSM) is susceptible to measurement 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 included along the same lines in a further, more refined specification of the RSM system.

7.1 Constructing the Specification

Figure 3 shows the RSM specification represented as an ArchiTRIO class diagram including the broadcast mechanism (class `Broadcast`), the vehicle's sensor (class `Sensor`) and broadcast receiver (class `ADS-B`), and the display (class `IntegratedDisplaySystem`). The specification, as created here, contains a mix of requirements and specification variables because, in our particular technique for verification, the link between the two types of variables—the set A—is contained within the specification. Also, the specification consists mostly of variables that change over time (denoted «`TD`»), but `airport_info_S`—which represents the information that a vehicle has about the airport—is static (denoted «`TI`») since we model the behavior of aircraft only when they are in (the vicinity of) one airport.

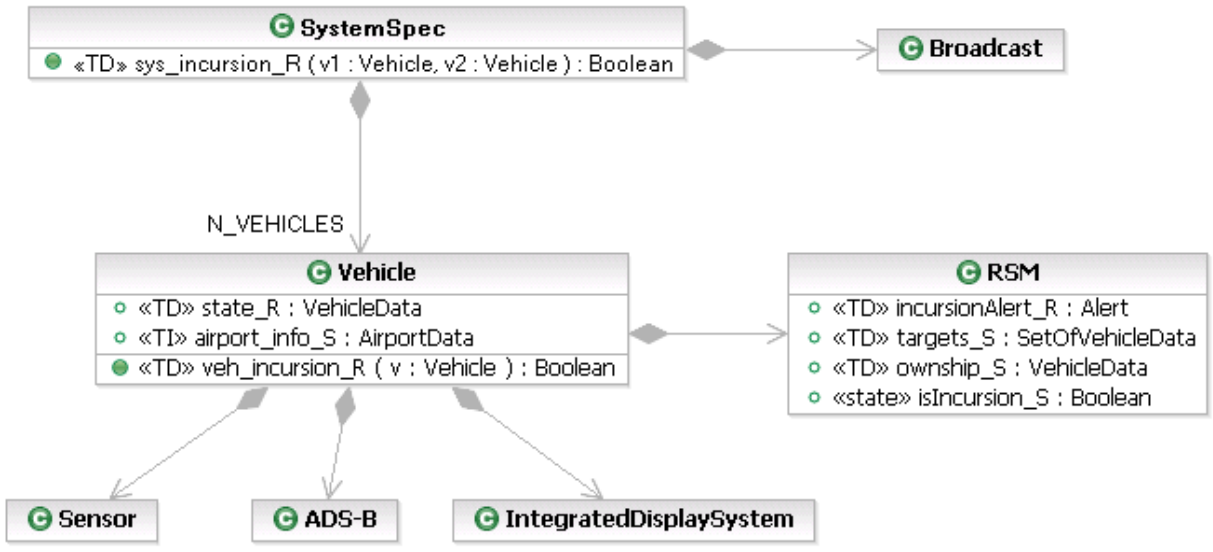

Figure 3 - ArchiTRIO classes representing the elements of the specification.

The structure diagram [OMG05] of Figure 4 shows that vehicles and the broadcast mechanism interact with each other; the boxes crossing the borders of classes (e.g. `out_world`)

represent UML2/ArchiTRIO ports, which define the interfaces and the dynamics of the interaction among classes (see [PRM05] for the syntax and semantics of ArchiTRIO elements).

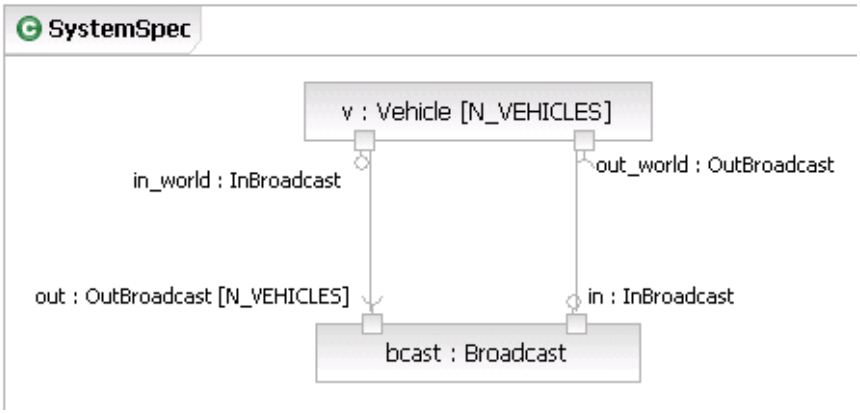

Figure 4 - High-level structure of the system.

In addition, the structure diagram of Figure 5 precisely defines how the different components of a vehicle are connected with each other (and with the environment).

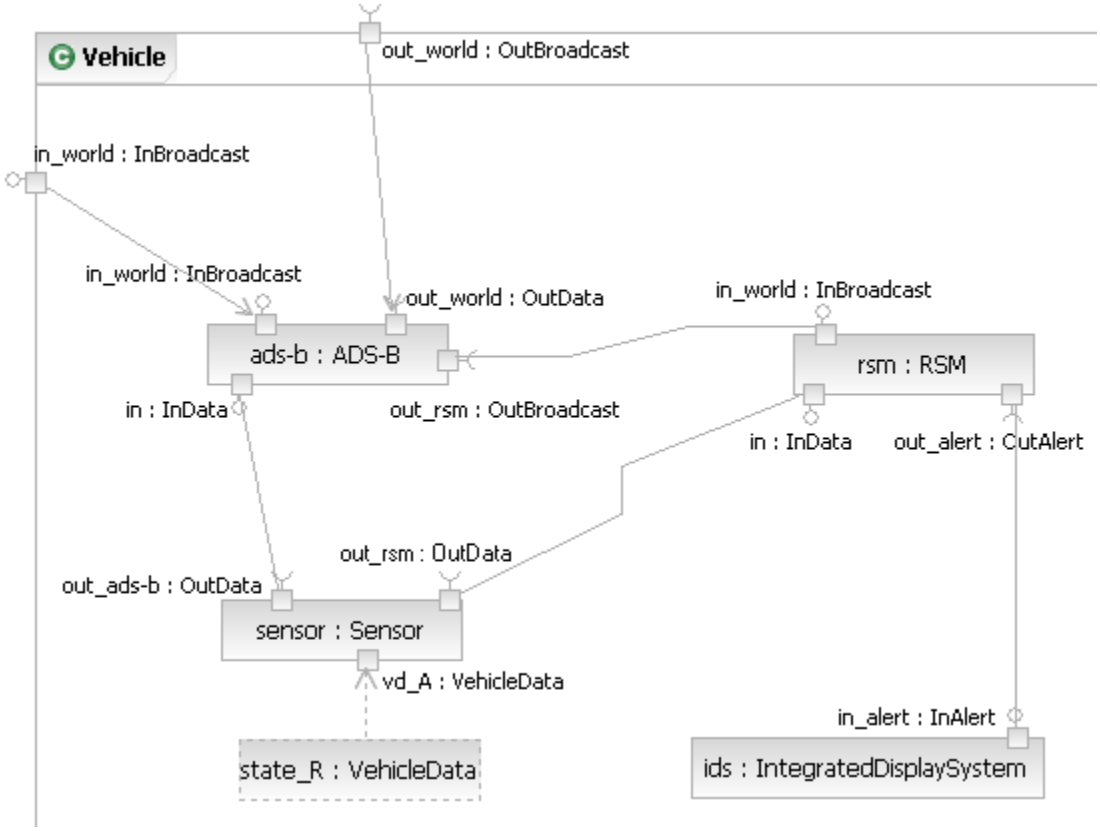

Figure 5 - Structure of class `Vehicle`.

The diagram of Figure 6, instead, defines port types `InData` and `OutData` (of which ports `in` of class `RSM` and `out_rsm` of class `Sensor`, respectively, are instances, as shown in Figure 5); more precisely, Figure 6 shows that ports of type `InData` *provide* interface `Incoming`, which offers operation `recData`, while ports of type `OutData` *require* the same interface (hence `OutData` is the dual of `InData`).

Notice that the diagrams of Figures 4−6 include solely elements that are in common between UML2.0 and ArchiTRIO; as mentioned in Section 4.1, such elements retain in ArchiTRIO the *same* meaning that they have in UML2.0 (hence, the diagrams of Figures 4−6 are in effect also UML2.0 diagrams), the only difference being that in ArchiTRIO such meaning is defined in a formal way.

Figure 6 - Example of definition of ports in ArchiTRIO.

As outlined at the beginning of the section, each of the new components can introduce a delay into the system between when the incursion occurs and when the system detects it. The bounds on these delays are members of the set A (Figure 1 and [SFRKM06]). We introduced formulas with each component to define a bound on the potential error that the component could introduce into the system.

Since formulas in A predicate about both world and system variables, the time model (i.e., world or machine time) they use is arbitrary, provided the relationship between the two models allows one to bridge the two time models to prove that the specification formulas (in machine time) imply the requirements formulas (in world time). From a practical viewpoint, we could choose one of two equivalent strategies:

- Express all formulas in A with reference to world time. Then, when carrying out the implication proof from specification to requirements, one has first of all to infer from the specification formulas other analogous formulas that refer to world time. Afterwards, all the remainder of the implication proof can be carried out in world time, in which the requirements are expressed.
- Express all formulas in A with reference to machine time. Then, when carrying out the implication proof one reasons completely in machine time, until some formulas, analogous to the requirements but referring to machine time, have been deduced. Finally, the implication proof is completed by inferring from the requirements, stated in machine time, to the "real" requirements, stated in to world time.

In our example we follow the second strategy: since we group formulas in A in the same components as the specification formulas, let us assume that formulas in A are interpreted over machine time, just like specification formulas. In our case, this will simplify the details that need to be expressed about the relationship between the two times, as the RSM requirements that we consider admit a particularly simple representation.

Let us provide an example of formulas in A. Each aircraft's sensors will estimate the real-world state values and then send these estimates to the RSM algorithm and to the broadcast system within `D_DS` machine time units of their being true of the aircraft. This 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_A` is the actual state of the vehicle (which corresponds to the value represented by item `state_R` of class `Vehicle`, as shown in Figure 5); and `d` denotes the vehicle's state:

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

This formula is included in the ArchiTRIO class for the `Sensor` specification component. Similar delay formulas are introduced for other time components at the appropriate places in the specification, filling out the set A for the RSM and distributing it where it can be easily validated against its corresponding element of the system design (such as the use of a particular sensor). The need to include explicit axioms linking requirements and specification is another advantage of formalizing system requirements: it forces domain experts and system designers to document assumptions that would often be left implicit, potentially leading to system flaws.

7.2 Design Assumptions

To avoid overconstraining the system design, the RSM specification includes a number of items whose exact value is not fixed: such are the various constants representing bounds on the delays we have described above. Figure 7 depicts these constants, together with the components of the system that introduce the delays. Recall that all these delays are expressed in machine time.

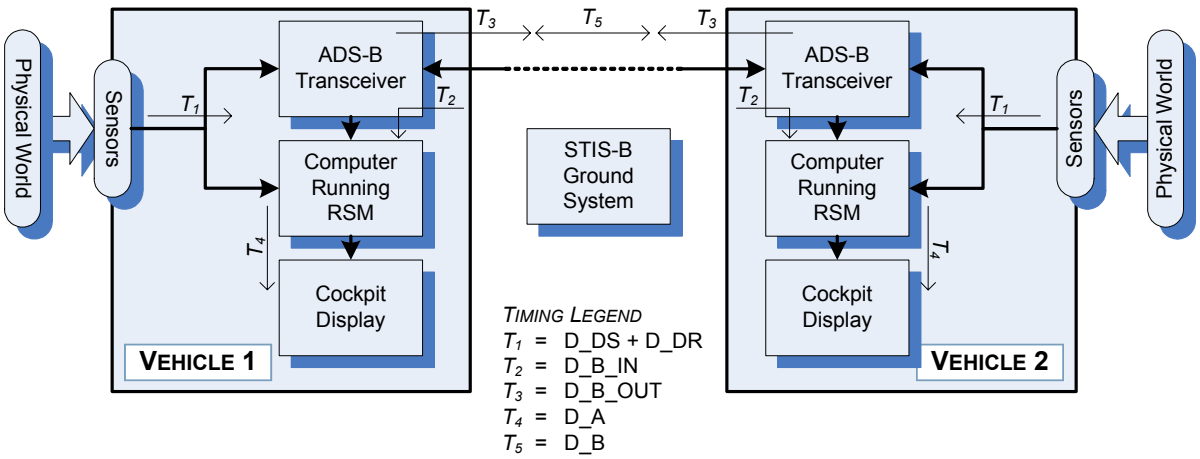

Figure 7 - Components and Delays in the RSM System

Most of the above delays are self-explanatory, but we will briefly address two more complex ones: `D_DS` and `D_SR`. `D_DS` is the maximum age that a sensor reading may have when that reading is transmitted to the system (as defined by formula `data_sent_def_A` of Section 7.1). `D_SR` is the

maximum period with which the readings are transmitted to the system. Combined, they mean that data received from sensors is at most `D_DS + D_SR` time units old. For simplicity, we assume that `D_DS >= D_SR`, and so hereafter refer to the combined delay as `2*D_DS`.

The RSM specification also introduces two new variables, `M` and `I_H`, which do not relate to the delays discussed above. We added these variables when we found that we were unable to verify the specification against the requirements. The problem that led to the introduction of the assumptions is discussed in Section 9; here, we explain the design assumptions that we had to make about the variables:[4]

```
M >= 2*D_DS + D_B_OUT + D_B + D_B_IN;                                    (DA1)

Lasts(veh_incursion_R(trg), I_H) &                                       (DA2)
0 < t1 < I_H & Futr(state_R = d_own & trg.state_R = d_1, t1) &
0 < t2 < I_H & Futr(state_R = d_2 & trg.state_R = d_targ, t2) &
airport_info_S.isIncursionState(d_own, d_1) &
airport_info_S.isIncursionState(d_2, d_targ)
-> airport_info_S.isIncursionState(d_own, d_targ);
```

The first formula documents an assumption about the value of the constant `M`. Another axiom, which we do not show here for brevity, describes how `M` characterizes the minimum time that the ownship vehicle keeps a sensed value in its memory, i.e. the local memory of the RSM will retain sensed values for at least `M` time units. . In order for the system to function properly, `M` must be long enough that, for every other vehicle in the system, there is some point at which the position measurement taken for that vehicle was taken within a small time bound (discussed below) of the position measurement stored for ownship in the RSM's memory. This must be true no matter how far apart the two transmissions occur; we can, however, assume that the transmission delays are within the time bounds we defined above. In practice, if `M` is at least as

[4] Recall that we assume an implicit universal quantification of all free variables.

large as the time bound `2*D_DS + D_B_OUT + D_B + D_B_IN`, then we can rely on the fact that such an instant exists.

The second formula documents our assumption that, if we observe two vehicles over a sufficiently short time interval, their states evolve in such a way that the `veh_incursion_R` predicate is "robustly" true (or false), where robust refers to a sort of closure property. The constant `I_H` (for "Incursion Holding") ensures that, if an incursion between ownship (*own*) and target (*targ*)[5] lasts for `I_H` time units, then any state `d_own` of the ownship and any state `d_targ` of the target occurring in the interval are such that `d_own` and `d_targ` correspond to an incursion (i.e. `airport_info_S.isIncursionState(d_own, d_targ)` holds). This property requires that the delay introduced by unsynchronized clocks on the different vehicles will not preclude the RSM from successfully detecting an incursion. We also introduce the assumption

```
I_H <= 2*D_DS                                                    (DA3)
```

which puts a bound on how short `I_H` must be, ensuring that the previous assumption on `I_H` is realistic.

All of the above specification variables must satisfy certain constraints, relating them to maximum allowable requirements delays, to ensure that those delays are met. These constraints are 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

---

[5] The formula belongs to class `Vehicle`. Therefore, the reference to the ownship vehicle is implicit, being the "this" instance. On the other hand, the target vehicle is explicitly mentioned in the formula through the variable `trg` of type `Vehicle`.

a lower-level design [FRMM06]. Two major assumptions we made, relating delays that are permissible from a requirements point of view and delays that must be bounded during system design, are:

```
L_INC >= 2*D_DS                                    (DA4)
```

```
D_INC >= M + D_A                                   (DA5)
```

These formulas can be assumed in verification activities undertaken before design is complete, even though their exact values might not be known.

7.3 Form of the Specification

The RSM specification has a structure similar to the requirements, and even a number of components that map directly to requirements components (for example the `Vehicle` and `RSM` classes, which appear in both the requirements and specification class diagrams of Figures 2 and 3). Direct mappings are helpful at the verification stage (described below), but they are not necessary—as seen by the broadcast mechanism, a component that exists only in the specification.

In addition, both the members of A and the specification design assumptions are documented in their corresponding 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));
```

7.4 Linking World Time and Machine Time

Let us finally describe how the world and machine time are related in our system formalization.

Recall that all the specification formulas are expressed in terms of machine time, as are the formulas in A. Therefore, we need to introduce statements that relate the timing of a world variable in machine time to the timing of the same variable in world time. In order to do that we ultimately need to relate the truth of some formula interpreted over machine time with that of some other formula interpreted over world time. Notice that such a relationship cannot be completely expressed in TRIO, as it requires to relate two different interpretations within the same formula.[6] Nonetheless, we can express it in a completely precise manner without resorting to TRIO notation.

Moreover, we think that expressing a detailed relationship between the two models of time would be too premature with respect to the high level of abstraction of our RSM specification. Therefore, we introduce an *ad hoc* link between the two models of time, which is sufficient to prove the requirements, but otherwise does not constrain any other aspect of the two time models. In a sense, this is the *weakest* assumption that one may make on the relationship between the two time models which is sufficient to guarantee meeting the requirements by the given specification. So, our assumption on machine time is two-fold:[7]

1. whenever the formula `Lasts(veh_incursion_R(v), L_INC)` holds in world time, then the same formula holds with respect to machine time;

---

[6] Technically, it requires a higher-order language. Although higher-order extensions of TRIO—which would make it possible to express formally the link between the two interpretations—have been provided [FMMPRS04], in this paper we refer to the "standard" TRIO language, which is first-order.

[7] These assumptions could be made arbitrarily more complex, and in particular one could introduce a further "error", such as a skew or a drift of the machine clock with respect to world time. For simplicity and minimality, we omit the discussion of such complications, which can however be handled by means of the same techniques.

2. whenever the formula `ex a (WithinF(rsm.incursionAlert_R = a, D_INC))` holds in machine time, then the same formula holds with respect to world time.

8 Verification of the Specification Against the Requirements

At this point, even in this relatively high-level model, the formalization is complex enough that automated analysis would be helpful in determining whether the specification implements the requirements. The ability to do this is a major advantage of the strategy of formalizing both requirements and specification; the more abstract the first formalization of the software, the more easily it can be validated by experts.

8.1 Linking Requirements and Specification

As noted in [SFRKM06], the major advantage of the A component of Figure 1 is that it provides the link between the requirements variables (i.e., world variables) and the specification variables (i.e., machine variables). The verification itself is given by the high-level picture S, A $\vdash$ R. The meaning of this statement is that the specification, coupled with the information about the link between world and machine variables, should imply that the requirements are satisfied. It was originally stated [Jac00] merely as an explanation of the underlying goal rather than as a particular step, since requirements were assumed to be informal and thus verification would not be possible in a formal way. With formalized requirements, this statement can be shown to hold for a given system using formal verification.

8.2 Translation to Verification Engine

In order to conduct the formal verification, the ArchiTRIO specification and requirements are first translated into the language of a verification engine. While ArchiTRIO itself does not mandate the use of a specific verification technique (be it model checking, or theorem proving, or even informal or semiformal techniques [MMM95,SMM00]) in this work we used the PVS

theorem prover [ORS92] as verification engine, for a number of reasons: its generality, which allows for example to leave details such as the exact values of constants undetermined (a crucial feature, for what was discussed in Section 7.2); our familiarity and previous experience with the tool; and the closeness of the core concepts of the PVS language with those that are at the basis of the semantics of ArchiTRIO.

Concerning this last point, let us remark that the PVS language consists of a typed higher-order logic [OS99]; similarly, the semantics of ArchiTRIO [PRM05] is given in terms of a higher-order variation of the TRIO language named HOT (Higher-Order TRIO [FMMPRS04]). As a consequence, translating ArchiTRIO formal models into the PVS language is fairly straightforward if one uses the HOT semantics of ArchiTRIO. For reasons of clarity and conciseness, in this article we will not show how the actual translation is defined; nonetheless, let us note that, while the translation is currently done by hand, it would be quite easy to fully automate it. Then, the translation enables the PVS theorem proving system to be used for formal verification activities on ArchiTRIO models.

Finally, let us also briefly remark that, being the PVS language a higher-order logic, it is even possible to express formally the link between the two time models of machine and world time, so that the whole verification process is completely tool supported and carried out in a formal way.

8.3 The Verification Process

Having translated our TRIO formulas into PVS axioms, we are able to use the PVS tool to construct a formal proof that S, A $\vdash$ R. Although we made certain simplifying assumptions about the system—documented in the previous sections—, the proof involved complex interactions and a number of details. In order to render the process feasible, we split the final goal into a set of steps whose composition yields the requirements theorem. Splitting the overall verification

burden into steps is useful not only in rendering the overall proof feasible, but also in making explicit the structure of the proof and its justifications. This allows one to gain more confidence in understanding the behavior of the formalized system artifacts, and to modularize the proof into independent—or only loosely related—parts. Indeed, in Section 9, we will highlight some crucial details of the specification that have been exposed during the verification process.

The requirement for the RSM is formalized by formula `incursion_detection_R` presented in Section 6.1. The verification basically consists in a proof of this formula, which is therefore the theorem to be proven.

Given this goal theorem, we constructed three intermediate lemmas which we could then compose to establish the theorem. We stated these lemmas in TRIO, in the class `Vehicle`, and proved them in PVS; the reader can intuitively see their derivations through the communication structure of the RSM structure diagrams (Figures 4−5). Recall that the proof references machine time until the very last passage, when the requirements in world time are shown to follow from the analogous requirement formula holding in machine time.

Lemma 1. The sensor of the ownship vehicle will deliver some data about the incursion to the RSM unit of the ownship vehicle within a specified time frame.

```
incursionstate_rsm_recv_own:
  Lasts(ex st1, st2 (sensor.vd_A = st1 & trg.sensor.vd_A = st2 &
                     airport_info_S.isIncursionState(st1, st2)), 2*D_DS)
  ->
  ex inc_st1, inc_st2 ( airport_info_S.isIncursionState(inc_st1, inc_st2) &
                WithinF(ex rsm_rD (rsm_rD.recv(inc_st1)), 2*D_DS) );
```

Lemma 1 addresses the delay between when a vehicle's sensor reads its data and when the RSM of that vehicle receives it; this "local" delay can be bounded by `2*D_DS`. We have included value `D_DS` twice, because one is the data liveness period (i.e., the maximum amount of time that

elapses between one sample and the next one), and one is for the transmission delay itself. If we had neglected to account for both these delays, the lemma would have been unprovable because the domain assumptions would have only implied the longer time bound. The ability to formally account for such details is one of the main strengths of requirements formalization.

Let us discuss a few other significant aspects exposed by Lemma 1. First of all, notice that the existential quantification on `st1`, `st2` in the left-hand side of the implication is within the scope of the `Lasts` operator. This means that the pair of states, one for each of the two vehicles, that constitute an incursion state does in general change over time: what holds throughout `2*D_DS` time units is the fact that the two states constitute an incursion, but within this constraint the two states may change.

Second, notice that the formula also presents an existential quantification on the right-hand side of the implication. In practice, this means that the state value `inc_st1` broadcast by the ownship within the specified time bound does not have to match any of the states that satisfy the existential quantification on the left-hand side of the implication. This aspect may be somewhat counterintuitive, as one may expect that the state value that makes the incursion predicate true should be the same as the state value broadcast by ownship. However, the presence of the existential quantification on the right-hand side yields a *weaker* formula, that is, one that can be satisfied by more implementations because it is less constraining. Thus, the lemma requires fewer formalized details to be proved.

This aspect also exposes a subtle intertwining between the formalization of requirements (and specification) and how the real system is built. The particular formalization of this lemma does not require the "same" state that constitutes the incursion to be the one which is sent) because we are implicitly assuming that the incursion alerting feature just has to convey *whether*

an incursion is underway. It does not need to carry any additional data about the incursion itself, since the alert consists simply of an aural and/or visual flag. If, instead, our application was required to provide—to the pilot of the vehicle—not only whether an incursion is underway but also details of the state of the other vehicle involved in the incursion (e.g., its speed or altitude), then the present formalization should be strengthened to include an explicit relation between the state value that satisfies the incursion predicate and the one which is actually sent out to the other vehicles.

We believe that this discussion shows quite clearly the important role that the formalization—and formal verification—of requirements may have in an application development.

Lemma 2. The sensor of the target will deliver some data about the incursion to the RSM of the ownship vehicle within a specified time frame.

```
in_world_to_rsm:
  Lasts(ex st1, st2 (sensor.vd_A = st1 & trg.sensor.vd_A = st2 &
                     airport_info_S.isIncursionState(st1, st2)), 2*D_DS)
  ->
  ex inc_st1, inc_st2 ( airport_info_S.isIncursionState(inc_st1, inc_st2) &
                WithinF(ex rsm_in_bD (rsm_in_bD.recv(inc_st2)),
                        2*D_DS + D_B_OUT + D_B + D_B_IN) );
```

Lemma 2 addresses the delay between when a vehicle's sensor reads its data and when the ownship system receives some data about that incursion situation; this "global" delay is bounded by the larger quantity `2*D_DS + D_B_OUT + D_B + D_B_IN`. The difference between this quantity and the quantity in Lemma 1 consists of the broadcast delays, since the data must go out of the target, through its ADSB system, to the broadcast subsystem; then the same data will go through the ownship ADSB system to its RSM.

Combining Lemmas 1 and 2, we can show that if there is a sufficiently long incursion, the

RSM of the ownship receives data about the incursion from both its sensor and the target vehicle, within the time bound `2*D_DS + D_B_OUT + D_B + D_B_IN`.

Next, we combine Lemmas 1 and 2, along with the design obligations about `M`, to prove Lemma 3. This lemma states that the predicate `isIncursion_S` is true in the RSM within `M`, whenever an incursion lasts for longer than `2*D_DS`:

Lemma 3. The RSM of the ownship detects an incursion within `M` time units whenever the incursion lasts at least as long as the two vehicles' sensors could take to detect an incursion.

```
rsm_detects_incursion:
  Lasts(ex st1, st2 (sensor.vd_A = st1 &
                     trg.sensor.vd_A = st2 &
                     airport_info_S.isIncursionState(st1, st2)), 2*D_DS)
  ->
  WithinF(rsm.isIncursion_S, M);
```

This lemma can be proved using the design obligations (DA1)-(DA3) about `M` and `I_H` mentioned above. In a nutshell, from Lemmas 1 and 2 we know that the ownship receives, within a given time span, data both about itself (from its RSM) and about the target (from the broadcast system). If an incursion lasts for `2*D_DS` time units, from assumption (DA3) about `I_H` it follows that the received data must be incursion data. Moreover, the received data are both stored in the RSM memory, given the role of the constant `M`. All in all, the two data can be matched to conclude that the predicate `isIncursion_S` holds.

The final step in the verification, which proves the requirements directly from Lemma 3, is straightforward. Assuming all the aforementioned design assumptions (DA1)-(DA5), a sufficiently long incursion triggers the `isIncursion_S` predicate to become true, which in turn triggers an alarm to be sent to the IDS module. The overall delay is determined by the constant `M` plus the additional delay `D_A` to trigger the alarm. Given the assumption (DA5) on the value of

D_INC, this assesses the validity of the requirement formula with respect to machine time. Finally, the stated link between world and machine time allows us to infer the validity of the same requirement formula with reference to world time, which corresponds to the validity of the "real" requirements.

9 Discovery of Requirements/Specification Disagreement

When we conducted the verification of the RSM specification against its requirements, we found that Lemma 3 could not be proved from our original specification. The problem lay in the fact that, in general, the sensors on two different aircraft are not synchronized. Let us assume, for instance, that the ownship's sensor sends its state d_own at time t_own, while the target's sensor sends its state d_targ at time t_targ. If an incursion is taking place, airport_info_S.isIncursionState(d_own, d_1) holds for some target's state d_1 (at time t_own), and airport_info_S.isIncursionState(d_2, d_targ) holds for some ownship's state d_2 (at time t_targ).

However, in general, we cannot conclude that the two collected data d_own and d_targ are such that airport_info_S.isIncursionState(d_own, d_targ) is true. As an example of this, consider two aircraft that are marginally in an incursion situation, moving parallel to each other. Assume one aircraft's sensor is operating slightly ahead of the other. In this case, because of the sensor delay, the perceived distance between the two aircraft is increased and so the incursion is not flagged (as shown in Figure 8). Because airport_info_S.isIncursionState(d_own, d_targ) must be true in order for the isIncursion_S state to be true, rsm_detects_incursion is not satisfied in this case.

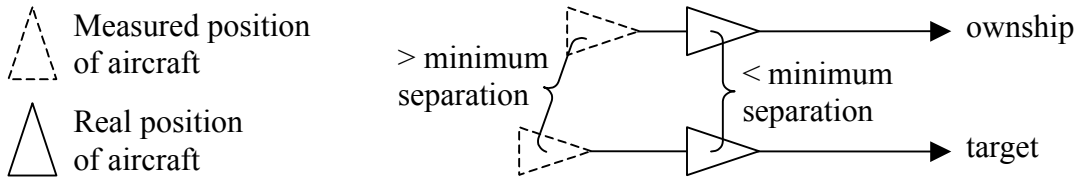

Figure 8 - Case where specification does not meet requirement

Unless we assume—erroneously—that the sensor of each vehicle sends its data continuously to the outside world (and thus there is no "information loss" and the RSM has all the possible data, including those that made `airport_info_S.isIncursionState` true), we cannot be sure that the relevant data is transmitted. We do not want to introduce a global synchronization mechanism, since dealing with asynchrony is one of the main aspects that is analyzable through our model.

As explained above, we introduced the design variables `M` and `I_H` to solve this problem, which was brought to our attention when we attempted to verify the specification against its requirements. The use of `I_H` in formulating a "continuity property" of the incursion predicate was presented in Section 7.2. This property of the incursion predicate—if two vehicles are in an incursion over a time interval, then small transmission delays within that time interval will not affect whether the two vehicles are calculated to be in an incursion—precludes the incursion scenario explained above.

Our discovery of this inconsistency shows one of the major advantages of formalizing requirements: interactions among system phenomena can be analyzed formally, and subtle properties of implementations that could lead to system failures can be uncovered. This is especially important in real-time systems, where those interactions are harder to assess because of the temporal component. The formalization was very helpful even in our relatively simple system; in more complex systems it could have a significant impact, even if the original time

bounds seem easy to achieve, because of accumulated delay at various points.

10 Related Work

The ArchiTRIO language and its predecessor TRIO have already been used to express and analyze properties involving heterogeneous elements, such as a controlled environment and a controlling machine (see, e.g., [CCCMMM99]), thus ArchiTRIO appeared as a good candidate to support the method illustrated in this paper to formalize requirements and to verify correctness of the design specification against them.

Needless to say, formalized requirements have been presented by others. Jackson gives small examples of formal requirements and their relationship to specification [Jac00]. Also, Jones et al. [JHJ06] use Duration Calculus [CHR91] in a problem frame setting [Jac00] to formalize both the problem (i.e., the requirement) and the solution (i.e., the specification) of a system that interacts with the physical world; however, they focus mostly on the issue of modeling system (i.e., environment and machine) dynamics, and they do not deal with the problem of verifying that the designed solution actually solves the problem.

Also with reference to Jackson's problem frames, Hall et al. deal with the problem of guaranteeing that the specification meets the requirements (i.e., that the solution matches the problem) in a formal way both in [HRJ05] and in [HRJ06]. However, in neither case do they tackle explicitly the problem of time modeling.

Hall and Rapanotti [HR03] introduce a notion of time in the Reference Model of Gunter el al. [GGJZ00] to clarify which system dynamics are relevant when determining whether the specification meets the requirements (i.e., in terms of the Reference Model, if W, S $\vdash$ R); they do not, however, provide a technique to actually check if W, S $\vdash$ R.

In the KAOS methodology [DvLF93] requirements can be formalized through the MTL

metric temporal logic [Koy90]. However, the emphasis in the KAOS methodology is on (goal) refinement, not on verification of the adequacy of a specification with respect to its requirements.

Chechik and Gannon [CG01] propose a technique to formally verify whether requirements expressed in the SCR notation [HPSK78, HJL96] are met by a program designed through a Program Design Language (PDL) [CG75]. On the other hand, the artifact that Chechik and Gannon refer to as "requirements" corresponds to our notion of "specification" [CG01]. Hence, their technique would be applied at a later stage in the system development process, and it is complementary rather than alternative to ours. Also, Chechik and Gannon [CG01] deal only in passing with timing requirements, which are instead a focus of the present work.

Finally, van Lamsweerde provides general overviews of issues and results in the domains of requirements engineering and formal specification research [vLa00a,vLa00b].

11 Conclusion

Whereas requirements are often considered an informal notion, in this paper we have shown that they can be formalized just as specification can. The formalization of requirements is made possible by building on a clear-cut distinction between the requirements artifact and the specification artifact; this distinction was one of the results in the companion paper [SFRKM06].

The major advantage of formalizing requirements is that one can bring the traditional techniques of formal analysis up to the abstract requirements/specification level. We demonstrated how to pursue this advantage in practice through the illustrative example of an airport incursion detection system. The focus of the example was on timing aspects, which have been dealt with through the formal notation ArchiTRIO [PRM05].

We tackled the crucial problem of formalizing explicitly the relationship between variables of the environment and variables of the system. In particular, we have also shown how to deal

with two separate notions of time, namely *world* time, i.e., physical, real, time, and *machine* time, i.e., time as perceived by an artifact that constitutes the system being developed.

The clear distinction between requirements and specification, and the formalization of both, has uncovered a subtle flaw in our formalization of the requirements and specification. This flaw was overcome by suitable explicit assumptions on the behavior of the environment. More generally, the process has greatly increased our confidence in understanding the behavior of the system and its interaction with the environment.

11.1 Future Work

Whereas this paper tackled both methodological and technical aspects, it focused on the former. Further developments on the technical side are therefore the object of future work. In particular, the world vs. machine time issue could be further investigated, possibly also in the direction toward actual implementations. Second, different verification and proof techniques tailored to the requirements verification problem will be investigated. Finally, the practical adoption of the whole formalization and verification process would greatly benefit from the development of a suitable, extensive tool support.

Acknowledgments

The authors thank Jon Hall, Michael Jackson and Bashar Nuseibeh for comments and useful suggestions of references and related works. 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

[Abr96] J. R. Abrial: *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[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.

[CG75] S.H. Caine and E.K. Gordon: “PDL: A Tool for Software Design”. In *Proceedings of the National Computer Conference*, 44:271-276, 1975.

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

[CG01] M. Chechik and J. D. Gannon: “Automatic Analysis of Consistency between Requirements and Designs”. *IEEE Transactions on Software Engineering*, 27(7): 651-672, 2001.

[DvLF93] A. Dardenne, A. van Lamsweerde and S. Fickas: “Goal-Directed Requirements Acquisition”. Science of Computer Programming. 20(1-2): 3-50, 1993.

[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.

[FMMR] C. A. Furia, D. Mandrioli, A. Morzenti, and M. Rossi: “Modeling time in computing: a taxonomy and a comparative survey”. In preparation.

[FR06] C. A. Furia and M. Rossi: “Integrating Discrete- and Continuous-Time Metric Temporal Logics Through Sampling”. In *Proceedings of the 4th International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS'06)*. Lecture Notes in Computer Science, 4202:215-229, 2006.

[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.

[GM01] Gargantini, A., and Morzenti, A: "Automated deductive requirements analysis of critical systems". *ACM Transactions on Software Engineering and Methodology* 10(3):255−307, 2001.

[HJL96] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw: "Automated Consistency Checking of Requirements Specifications". *ACM Transactions on Software Engineering and Methodology*, 5(3):231-261, 1996.

[HM96] C.L. Heitmeyer, D. Mandrioli (editors): *Formal Methods for Real-Time Computing*. Volume 5 of Trends in Software, John Wiley & Sons, 1996.

[HPSK78] K. Heninger, D. L. Parnas, J. E. Shore and J. W. Kallander: "Software requirements for the A-7E aircraft". Technical Report 3876, Naval Research Laboratory, Washington, D.C., 1978.

[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.

[HRJ06] J. G. Hall, L. Rapanotti, and M. Jackson: "Problem Oriented Software Engineering". Technical Report 2006/10, Centre for Research in Computing, The Open University, 2006.

[HS06] T. A. Henzinger and J. Sifakis: "The embedded system design challenge". In *Proceedings of the 14th International Symposium on Formal Methods (FM'06)*. Lecture Notes in

Computer Science 4085:1-15, 2006

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.

[Jon90] C. B. Jones, *Systematic Software using VDM*, Second Edition. Prentice Hall, 1990.

[Koy90] R. Koymans: "Specifying real-time properties with metric temporal logic". *Real-Time Systems*, 2(4):255–299, 1990.

[Liu00] J. W. Liu: *Real-Time Systems*. Prentice Hall, 2000

[MMM95] D. Mandrioli, S. Morasca, and A. Morzenti: "Generating Test Cases for Real-Time Systems from Logic Specifications". *ACM Transactions on Computer Systems,* 13(4): 365-398, 1995.

[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.

[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.

SFRKM06] E. A. Strunk, C. A. Furia, M. Rossi, J. C. Knight, and D. Mandrioli: "The

Engineering Roles of Requirements and Specification". Technical Report CS-2006-21, University of Virginia, 2006.

[SLMR05] J. A. Stankovic, I. Lee, A. Mok, and R. Rajkumar: "Opportunities and obligations for physical computing systems". *IEEE Computer*, 38(11):23-31, 2005.

[SMM00] P. San Pietro, A. Morzenti, and S. Morasca: "Generation of Execution Sequences for Modular Time Critical Systems". *IEEE Transactions of Software Engineering* 26(2):128-149, 2000.

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

[vLa00a] A. van Lamsweerde: "Requirements engineering in the year 00: a research perspective". *Proceedings of ICSE 2000*: 5-19, 2000.

[vLa00b] A. van Lamsweerde: "Formal specification: a roadmap". *Proceedings of ICSE - Future of Software Engineering Track 2000*: 147-159, 2000