Modeling the Environment in Software-Intensive Systems

Carlo A. Furia, Matteo Rossi, and Dino Mandrioli
Dipartimento di Elettronica e Informazione, Politecnico di Milano
{furia, rossi, mandrioli}@elet.polimi.it

Abstract

In this paper we argue that the modeling activity in the development of software-intensive systems should formalize as much as possible of the environment in which the application being developed operates. We also show that a rich formal model of the environment helps developers clearly state requirements that might typically be considered intrinsically informal (or non-formalizable in general). To illustrate this point, we show how a requirement for "orderly safe traffic" in a traffic system can be modeled, and we briefly discuss the benefits thereof.

1. Introduction

In today's engineering practice, we witness the ever increasing pervasiveness of software in a wide and heterogeneous variety of systems — where it was absent in the past. From a software engineering viewpoint, a natural consequence of this fact is the growing need to understand modeling of such *software-intensive systems*.

A software-intensive system (SIS) is a heterogeneous system whose software components are entangled with — and thus deeply interact with — other non-software components, such as mechanical parts, chemical processes, or even social organizations. The non-software components have in common their being part of the *physical world*. We denote by the term *environment* the non-software components together with the physical world to which they belong. Therefore, a SIS can be defined as a system with software components that interact with an external environment. Embedded systems constitute a very large part of such class of systems.

When it comes to modeling (and analyzing) a SIS, the central role of the environment constitutes the main concern of the software engineer. More precisely, there are at least two broad sets of properties of the environment that have to be modeled in a SIS: indicative properties and optative properties [4]. Indicative properties constitute a model of the physical world as it is; optative properties are instead properties that we would like to hold in the environment, as a consequence of the functionality (in a broad sense) of the whole system we build. Optative properties of the environment constitute the *requirements* of the system [4]. Therefore, in a SIS the interaction of the software components with the environment should meet the requirements.

Notice that, although a conspicuous number of SIS can be considered as controlled systems, the (more or less) traditional modeling techniques for control systems are not enough to fulfill all the modeling needs of a SIS. In fact, the difficulty of modeling a SIS lies precisely in the tight interaction of two traditionally distinct domains. Namely, one has to find ways to join software modeling techniques with physical modeling paradigms, without giving up the peculiarities of either.

In other words, analyzing the correctness of a SIS requires an accurate model: (1) of the environment; (2) of the software system; and (3) of their interaction. The optative part of the environment's model constitutes the requirements, whereas the software system model, at the highest level of abstraction, constitutes its specification. Then, verifying the system amounts to proving that the specification entails the requirements, with the given assumptions about the interaction between software and environment.

Traditionally, requirements engineering has been the realm of informal modeling and reasoning. To some extent, it is inevitable that some parts of the requirements may be given only an informal characterization, as they ultimately reside in customers' expectations about what the system should achieve. Nonetheless, it is increasingly argued that formalization can (and should) play a role even at the requirements level, and thus in dealing with properties of the environment [5].

However, how deep the *formal* modeling of environmental properties should (or can) be is still a debated issue. Even if "a formalized requirement is always incomplete" [5], how much effort and accuracy should be

29th International Conference on Software Engineering Workshops(ICSEW'07)
0-7695-2830-9/07 $20.00 © 2007 IEEE

invested in introducing the formalization of a significant part of a SIS's requirements (and environment)?

In this paper, we argue that *formalization can be pushed very deep in the environment domain*. We also demonstrate that this is not only possible, but also desirable, as it brings several advantages and capabilities in the analysis of the modeled system. In fact, the boundary between what can be formalized and what cannot is often fuzzy and application-dependent. Therefore, attempts at formalizing aspects that are "deeper in the environment" [5] of a SIS often result in a better understanding of the system and of the boundaries themselves. This usually offers the possibility of achieving a better system design, a more accurate verification process, and, more generally, a higher confidence in the dependability of the system that is built.

Some recent work to which we contributed [7, 1] has provided, among other things, additional evidence to these claims. In this paper we develop the theme further, with particular focus on SIS, through an example (a traffic system), shown in Section 3.

2. Pushing Formalization Deep in the Environment

In order to deepen the level of formality in modeling a SIS, two key "ingredients" should be available: a meta-model of the aspects involved and a suitable formal notation.

2.1. Meta-Model

A meta-model lays out precise definitions for the three macro components of a SIS that we have outlined above; namely: the environment (and its requirements), the software system (and its specification), and the interface between the two. A well-understood meta-model is very important to guide the formalization of the system, and even to set the boundaries between what is formalized and what is not. In this paper, we will refer to the meta-model that we have recently proposed in [7], which enhances Gunter et al.'s well-known reference model [2], and which we omit for reasons of brevity.

2.2. Formal Notation

The other basic ingredient for a deep formalization of a SIS is a suitable formal notation.

As we have outlined in Section 1, SISs are characterized by the heterogeneity of their components. Therefore, the formal notation should be very flexible, rich, and expressive, so that it can be used to describe a variety of aspects of the system: the diverse components, their dynamics (i.e., how they evolve over time), the data they exchange, etc. As SISs often have real-time requirements, the notation should permit to easily describe a rich set of temporal features. Finally, the notation should deal with the description of modules and their composition in a natural and powerful way, since SISs are inherently modular (and their analysis benefits greatly from a modular model).

ArchiTRIO [6] is a UML-compatible formal language which, at its core, has a very expressive and general metric temporal logic that permits to describe the dynamics and temporal constraints of the phenomena of interest (be they in the environment or in the machine domain). It is endowed with the modular constructs and encapsulation mechanisms of UML such as class and interface, which retain the same meaning and graphical representation they have in UML.

We will use the ArchiTRIO language for the development of the example of Section 3. While we will explain the features of ArchiTRIO when appropriate, a presentation of its syntax and semantics is beyond the scope of the present article, and is omitted; the interested reader can refer to [6] for further details about the notation. Of course, other notations with similar features could be used to pursue the same approach.

3. Example: Traffic System

In [5], Jackson argues that:

> "Although a physical and human environment is a nonformal domain, the engineering task of developing the system must rely—somewhat as in established branches of engineering—on formalized descriptions of the physical world and on reasoning about those descriptions. It's possible to bring these formalizations and the associated reasoning within the program specification's purview, and hence within the scope of some formal verification tools and techniques."

Our point of view is similar to the one presented by Jackson, but with some differences. We maintain that it is important, for the correct development of a SIS, to provide a mathematical (i.e., formal) model of the environment that is as accurate as possible without it becoming unwieldy. Also, we argue that such a model need not exist only in relation to the application's specification; rather, as discussed also in [7], it should be developed separately (i.e., using only environment-

29th International Conference on Software Engineering Workshops(ICSEW'07)
0-7695-2830-9/07 $20.00 © 2007 IEEE

IEEE COMPUTER SOCIETY

specific variables), and then "connected" to the specification through a suitable set of axioms. Finally, we hold that the verification activity on the formal model of the environment should begin as soon as possible, possibly (if not preferably) before the latter has been related to the system's specification.

In this section, we illustrate our approach and point of view on modeling through a common example taken from [5], a traffic system. The goal of the traffic system is to ensure "orderly safe traffic". Let us formally define it through the ArchiTRIO formal language [6].

As mentioned in [5], the requirement of such a system is "deep in the environment", so that a model of a number of physical elements (roads, vehicles, pedestrians, etc.) must be introduced before one can formulate it. Figure 1 shows a fragment of a UML class diagram (which is also an ArchiTRIO class diagram) representing the physical elements that are necessary to formulate the requirement. As the figure shows, we focus our attention on a single intersection, which is the terminus of N_ROADS roads, where N_ROADS is a parameter of the intersection (a road that continues after the intersection is represented through two separate RoadEnds). At every road end vehicles and pedestrians arriving at the intersection can accumulate in queues; both the queue of vehicles and the queue of pedestrians are represented as instances of the generic class DynamicQueue: whenever a new vehicle v (resp. pedestrian p) arrives at the intersection, this is represented through an occurrence of event $\mathrm{arrive}(v)$ (resp. $\mathrm{arrive}(p)$) in QueueOfVehicles (resp. QueueOfPedestrians); similarly, event $\mathrm{depart}(v)$ (resp. $\mathrm{depart}(p)$) represents a vehicle (resp. pedestrian) leaving the intersection.

3.1. Environment Formalization

With a formal language as expressive as ArchiTRIO one can formally define the behavior of the queues informally described above. In fact, by exploiting the object-oriented features of ArchiTRIO, this can be achieved simply by adding formulas to class DynamicQueue, of which both QueueOfVehicles and QueueOfPedestrians are instances.

For example, the following formula of class DynamicQueue defines how the state s of a queue evolves over time whenever a new arrival or departure occurs.

$$
\begin{array}{l}
\forall q : \mathsf{Queue[T]}(\neg q.\mathrm{isEmpty} \Longrightarrow \\
(\quad \mathrm{s} = q \\
\qquad \Longleftrightarrow \\
\quad \mathrm{Since}(\nexists e_1 : \mathsf{T}(\mathrm{arrive}(e_1) \vee \mathrm{depart}(e_1)), \\
\qquad\quad \exists e_2 : \mathsf{T}, q' : \mathsf{Queue[T]}(\\
\qquad\qquad (\mathrm{arrive}(e_2) \wedge \mathrm{s} = q' \wedge q = q'.\mathrm{enqueue}(e_2)) \vee \\
\qquad\qquad (\mathrm{depart}(e_2) \wedge \mathrm{s} = q' \wedge q = q'.\mathrm{dequeue}) \;))))
\end{array}
$$

More precisely, the formula defines that, in any instant, the state of the queue is q (with q a value of generic type Queue, which will be further described below), and there is at least one queued "actor" (i.e. an element of a generic type T) if and only if there is an element e_2 that either previously arrived (and queued up) or departed (and left the queue), and noone has arrived nor left since.

For simplicity, we could also specify that one can leave the queue only if he/she is at its head (which, for example, in the case of a queue of vehicles represents the fact that overtaking a vehicle at the front of the queue is not allowed, nor one can make a U-turn and leave the queue). This would be described by the following formula (also of class DynamicQueue):

$$\forall e : \mathsf{T}(\mathrm{depart}(e) \Rightarrow e = \mathrm{s.head})$$

The value of state s of a DynamicQueue at a certain instant t describes what elements are queued at t, and in what order. Since elements enter and leave the queue in a First-In-First-Out policy, it is most natural to model the possible values of s through a classic "queue data type", such as for example the one defined in [3].[1] Hence, the value of state s is defined to be of generic type Queue. Type Queue can also be formally defined, in a manner similar to that of [3]. For example, the following formula of class Queue defines (in a "classic" fashion) what element is the head of a queue (which essentially states the FIFO nature of Queue):

$$
\begin{array}{l}
\forall q : \mathsf{Queue[T]}, e, e' : \mathsf{T}(\\
\quad (q.\mathrm{enqueue}(e)).\mathrm{head} = e' \\
\qquad \Longleftrightarrow \\
\quad (q.\mathrm{isEmpty} \wedge e' = e) \vee (\neg q.\mathrm{isEmpty} \wedge e' = q.\mathrm{head}))
\end{array}
$$

where $q.\mathrm{enqueue}(e)$ is the queue (say, q') obtained by appending element e (of generic type T) to queue q, hence $(q.\mathrm{enqueue}(e)).\mathrm{head}$ is the head of q'.

3.2. Requirements Formalization

After having introduced the elements of Figure 1, we can now state a possible requirement of "orderly safe traffic". Informally, we state that the traffic is orderly and safe if vehicles and pedestrians arriving at the intersection from directions that are "incompatible" with each other (e.g., perpendicular ones) enter the intersection with sufficient delays from each other.

To formalize this requirement, we first introduce a predicate, $\mathrm{conflicting}(r_1, r_2)$, in class Intersection (where r_1 and r_2 are RoadEnds of the Intersection itself); intuitively, predicate $\mathrm{conflicting}$ describes which

[1] Note that there is no notion of "implementation" or "programming" here: We use the queue data type only as a device to *model* information, without any reference to any programming activity.

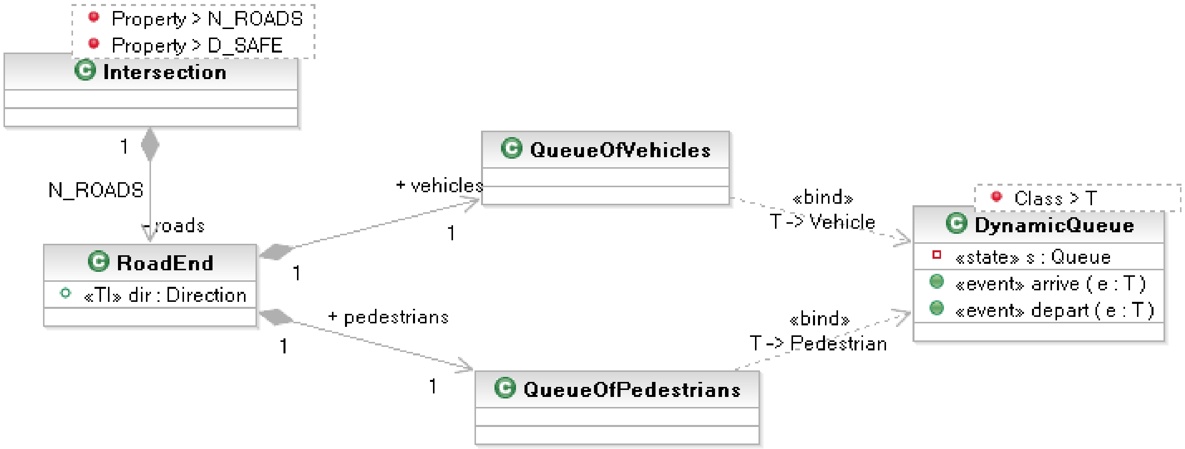

Figure 1. Elements of the environment of the traffic system.

roads are "incompatible", and it is true for all those pairs of roads from which vehicles (or pedestrians) cannot flow into the intersection at the same time. Finally, the requirement above can be represented by the following ArchiTRIO formula of class Intersection:

OrderlySafeTrafficReq:

$$
\begin{array}{l}
\forall r_1, r_2 : \mathsf{RoadEnd}(\\
\quad \text{conflicting}(r_1, r_2) \wedge \\
\quad \exists v_1 : \mathsf{Vehicle}(r_1.\text{vehicles}.\text{depart}(v_1)) \\
\qquad \Longrightarrow \\
\quad \nexists v_2 : \mathsf{Vehicle}(\text{Within}(r_2.\text{vehicles}.\text{depart}(v_2), \mathtt{D\_SAFE})) \wedge \\
\quad \nexists p : \mathsf{Pedestrian}(\text{Within}(r_2.\text{pedestrians}.\text{depart}(p), \mathtt{D\_SAFE})))
\end{array}
$$

Formula **OrderlySafeTrafficReq** states that, if r_1 and r_2 are two conflicting roads of the intersection and a vehicle enters the intersection from road r_1, then no vehicles and no pedestrians enter (or have entered) the intersection from road r_2 within D_SAFE time units from the current instant. Note that, in this model, D_SAFE is a constant delay that is a parameter of class Intersection; however, in a different (and a little more sophisticated) model it could depend on the speed of the vehicle, its direction, etc.

3.3. Discussion

Let us immediately note that the notion of "orderly safe traffic" stated above is by no means the only possible one (it is probably not the most accurate, either). For example, one may employ different notions, which take into account not only the road from which vehicles or pedestrians come, but also the one they wish to take; also, another possible definition of "orderly safe traffic" could be "the absence of accidents". Such notions could be formalized in ways that are similar to the one shown above, provided suitable additional elements are introduced in the model (e.g., the notions of "next road" and "position" for vehicles and pedestrians).

We argue that the mere exercise of precisely expressing the vague and imprecise notion of what constitutes "orderly and safe traffic" is a necessary step when one sets out to design a traffic control system. In fact, different notions of "orderly and safe traffic" might entail different solutions to the problem. For example, the requirement above is probably best satisfied by employing traffic lights; on the other hand, a requirement that asked for "minimum separation" between cars entering the intersection might be satisfied through a roundabout, and so on. Even this simple example shows that deepening the formalization of requirements sheds early light on the design decisions to be taken (later).

After the requirement has been precisely stated, the next design decision is how to meet it. As hinted above, requirement **OrderlySafeTrafficReq** does not mention the use of traffic lights; in fact the choice of employing traffic lights to ensure "orderly and safe traffic" is already a design decision (one taken in the early stages of the development process, but still a design decision). If, however, it is decided to use traffic lights, from the modeling point of view the next step would be to introduce the elements on which the specification will be built (i.e., as remarked in [7], the machine-domain variables). Following the lead of [7], then, one would introduce new elements such as light units, traffic sensors, system controller, and build a suitable set of axioms that describe how these elements are related to the physical components of Section 3.

While such an exercise is outside the scope of this article, let us hint at how it could be proved that the designed system actually satisfies the requirement of Section 3. If, for example, we assumed "good behavior" by pedestrians and drivers (which could be formalized by formulas stating that "a vehicle will not enter the intersection with a red light"), then a control system which ensures that no green lights are on at the same time on conflicting roads and that periodically all lights are red for a certain amount of time (which will depend in some way from D_SAFE) could be formally shown to guarantee the requirement as a logical consequence of such properties, and of the indicative properties of the environment, e.g., the formal description of the queues (see [1] for further details on this issue). Finally, let us note that proving properties in a deductive fashion is not a requisite in our approach; utility requirements stating a desired average traffic of vehicles through the intersection could be analyzed using different mathematical tools such as, for example, Markov chains.

4. Conclusion

In this paper we argued through a simple common example that in software-intensive systems great care should be given to the formal modeling of the environment with which the application interacts. Also, we showed how formal notations can be "pushed deep in the environment" and used to describe not only the software parts of the application, but also the physical elements which influence its behavior. This allowed us to formally express an "orderly safe traffic" requirement that might typically be considered inherently informal.

Acknowledgments

The authors thank Elisabeth Strunk and John Knight for the fruitful discussions and collaboration, which shaped several of the views expressed in this article.

References

[1] C. A. Furia, M. Rossi, E. A. Strunk, D. Mandrioli, and J. C. Knight. Raising formal methods to the requirements level. Technical Report 2006.64, Dipartimento di Elettronica e Informazione, Politecnico di Milano, November 2006. Also: Technical Report CS-2006-24, Department of Computer Science, University of Virginia.

[2] C. A. Gunter, E. L. Gunter, M. Jackson, and P. Zave. A reference model for requirements and specifications. *IEEE Software*, 17(3):37–43, 2000.

[3] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[4] M. Jackson. *Software Requirements and Specifications*. Addison-Wesley, 1995.

[5] M. Jackson. What can we expect from program verification? *IEEE Computer*, 39(10):65–71, 2006.

[6] M. Pradella, M. Rossi, and D. Mandrioli. ArchiTRIO: A UML-compatible language for architectural description and its formal semantics. In *Proceedings of FORTE 2005*, volume 3731 of *Lecture Notes in Computer Science*, pages 381–395. Springer-Verlag, 2005.

[7] 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, Department of Computer Science, University of Virginia, October 2006. Also: Technical Report 2006.61, Dipartimento di Elettronica e Informazione, Politecnico di Milano.

IEEE COMPUTER SOCIETY