Techniques for Reactive System Design: The Tools in TRADE

Roel J. Wieringa and David N. Jansen[*]

Department of Computer Science, University of Twente, P.O. Box 217, 7500 AE, The Netherlands
{roelw, dnjansen}@cs.utwente.nl

Abstract. Reactive systems are systems whose purpose is to maintain a certain desirable state of affairs in their environment, and include information systems, groupware, workflow systems, and control software. The current generation of information system design methods cannot cope with the high demands that originate from mission-critical application, geographic distribution, and a mix of data-intensive, behavior-intensive and communication-intensive properties of many modern reactive systems. We define an approach to designing reactive software systems that deals with these dimensions by incorporating elements from various information system and software design techniques and extending this with formal specification techniques, in particular with model checking. We illustrate our approach with a smart card application and show how informal techniques can be combined with model checking.

1 Introduction

The past ten years have shown an explosion of different types of information technology in which the classical distinction between information systems and control software has disappeared. In addition to the data-intensive applications like order administrations, and control-intensive applications like production control, there is now widespread use of email, office agendas, shared workspaces, workflow management, enterprise resource planning (ERP), electronic data interchange (EDI), and internet-based ecommerce applications. In these applications, we see a varying mix of complexity along the three major dimensions of functional software properties: data, behavior, and communication. (That these are the three important dimensions of functional software properties is argued elsewhere [13].)

A fourth dimension has emerged as important as well: geographical distribution. For example, classical information systems may be distributed over several sites, and they may be connected to classical production control systems in a complex network of applications that may even cross organizational boundaries.

A fifth dimension relevant to software is the degree to which it is mission critical. This is not a property of the software as such but of the way it is used by an organization. We now have many companies large parts of which basically *are*

[*] Partially supported by NWO-SION, project number 612-323-419.

software systems, operated by a few employees. This has long been the case in the finance business but it is now also the case for application service providers, supply chain integration, and business-to-consumer electronic commerce. When these applications fail, their businesses lose money by the minute. As a consequence, there must be ample attention to reliability, security, safety and other mission-critical attributes. We claim that formal techniques have a role to play in this area and later in this paper we argue how this can be done.

The current generation of functional and object-oriented methods do not suffice to deal with all of these dimensions. Structured techniques [11,15] tend to spaghetti-like data flow diagrams with a sloppy semantics, and object-oriented techniques have evolved into a Unified Modeling Language (UML) whose complexity is not motivated by the complexity of the systems to be designed, but by the number of stakeholders involved in defining the notation [12,10]. In addition, the complexity of the UML, as well as the complexity of the process in which the UML is defined, leads to a continuous stream of revisions and an incomplete and ambiguous semantics.

In this paper we propose a simple approach that picks the elements from structured and object-oriented approaches that have turned out in practice to be very useful, and extends this with a formal specification approach to deal the increasing need for reliability and precision. We claim that the resulting approach, called TRADE (Techniques for Requirements and Architecture DEsign) is useful bag of tools to use when designing complex information systems.

The unifying view that we present starts from the concept of a reactive system, introduced by Harel and Pnueli [4] 15 years ago. This is explained in section 2. Section 3 defines our mix of structured and object-oriented techniques and discusses how formal techniques can be combined with informal techniques. Appendix 4 illustrates out claim by an example specification of a smart card application, and of two properties that we model checked in our specification. Details about the techniques and guidelines are given elsewhere [3,6,14].

2 Reactive Systems

A reactive system is a system that, when switched on, is able to respond to stimuli as and when they occur, in a way that enforces or enables desired behavior in the environment [4,7]. Stimuli may arrive at arbitrary times and the response must be produced soon enough for the desirable effect to take place. Somewhere in the environment, *events* occur, that cause *stimuli* at the system interface. The system *responds* based on an internal model that it maintains of its environment, and the response leads to, or enables, a desired *action* in the environment. Examples of reactive systems are information systems, workflow management systems, email systems, systems for video conferencing, shared office agenda systems, chat boxes, group decision support systems, process control software, embedded software and game software.

Reactive systems are to be distinguished from *transformational systems*, which are systems that, when switched on, accept input, perform a terminating computation, produce output, and then turn themselves off. Examples of transformational systems are compilers, assemblers, expert systems, numerical routines, statistical procedures, etc. A transformational system can be viewed as computing single, isolated stimulus-response pairs. Transformational systems have no relevant internal state that survives a single stimulus-response pair. To specify a transformational system, you must specify a terminating algorithm, whereas to specify a reactive system, you must specify stimulus-response behavior. Both kinds of system must be switched on before used, but when a reactive system switches itself off this is because something wrong has happened. For a transformational system the opposite is true: when it does *not* switch itself off, something is wrong.

3 Requirements Specification for Reactive Systems

3.1 Functions

Reactive systems exist to provide services to their environment. They provide these services by responding to stimuli. We define a *function* of a system as any service delivered by the system to its environment. A function delivers value to its environment. It is the ability to deliver value to its environment that motivates someone to pay money for the system.

To specify a function, we must at least specify what value is delivered by the function and when it is delivered, i.e. which event triggers it. Figure 6 gives an example. In our view it is not a good idea to specify detailed scenarios for functions. These scenarios obscure the view of what the function is for (which value is delivered). They also mix requirements (what do we need the system for) with architecture (what are the high-level components that will deliver these services). As we illustrate in the appendix, a precise architecture description of the system will include a detailed specification of the behavior of the system.

3.2 The Environment

The interactions of a software system always consist of exchanges of symbol occurrences (e.g. data items, event occurrences) with the environment. A truly implementation-independent description of the data manipulated by the system restricts itself to the symbol occurrences that cross the external interface shown in the context diagram. These symbol occurrences have a *meaning,* which must be understood by the designers and users in order to understand the behavior of the system. The subject of these exchanges is called the *subject domain* of the interactions. For example, the subject domain of our ticket selling example system in the appendix consists of tickets, routes, etc. These are the entities about which the system communicates with its environment. The subject domain is often modeled by an entity model, such as illustrated in Fig. 9.

In addition to the subject domain, the environment of the system consists of a *connection domain*, which consists of entities that can observe the subject

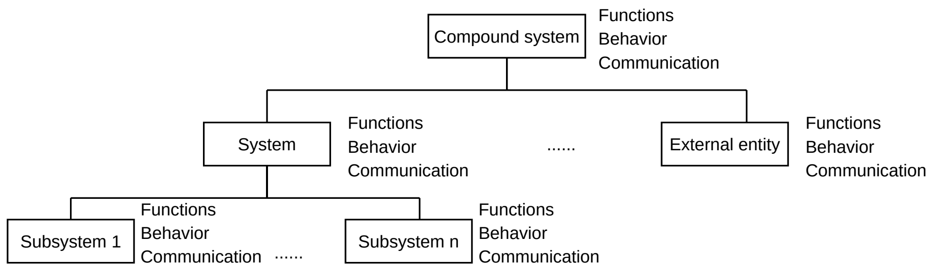

Fig. 1. Aggregation hierarchy and system aspects.

domain and provide stimuli to the system, and entities that can act on the subject domain based on the responses of the system. In control software, these entities are called sensors and actuators, but people can play these roles too.

A third element of the environment is the *implementation platform,* which is the collection of programmable entities that will contain the software. When software engineers talk about the software environment, they often mean the implementation platform. When information engineers talk about the environment, they usually mean the subject and connection domains.

3.3 Requirements and Architecture

We define a *requirement* as a desired property of the system and an *architecture* as the way components are put together in order to realize these desired properties. The architecture dimension introduces an aggregation hierarchy, in which components at a lower level jointly realize properties of a component at a higher level.

In our view, requirements are not restricted to external properties. A requirement of a system is just any desired property of the system, be it a desired function or a desire that the software be executable on a certain implementation platform. An important kind of requirement is of course the *functional* requirement, which is basically a description of the desired system functions. In addition to the value delivered to the environment, there are at least two important aspects of software system functionality that usually have to be specified, namely *behavior,* which is the ordering of stimulus-response pairs in time, and *communication,* which is the ordering of stimulus-response pairs in "space", i.e. the communication links that connect the stimuli and responses with events and actions in the subject domain. These aspects are repeated at every aggregation level. So for each part of the system, we can ask what its desired functions, behavior and communication properties are, all the way down to individual software objects (Fig. 1).

A software system has many other properties, including safety, security and reliability properties etc. For some of these properties, formal techniques are needed to show their presence or absence. We give an example in the appendix.

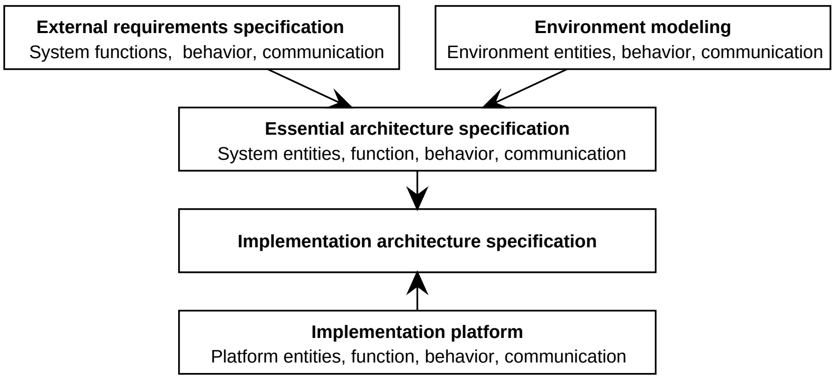
External requirements specification
System functions, behavior, communication
Environment modeling
Environment entities, behavior, communication
Essential architecture specification
System entities, function, behavior, communication
Implementation architecture specification
Implementation platform
Platform entities, function, behavior, communication

Fig. 2. The TRADE approach.

3.4 Design Approach

In this paper, by *design* we mean an activity in which the desired properties of a product are decided. Design is making decisions, and it results in a specification that documents these decisions. This use of the term agrees with that in other branches of manufacturing, where design is followed by planning the production process, manufacturing the components, assembling the components and distribution and sales.

This means that specifying requirements in our terminology is a design activity! Requirements specification is solution specification; the problem to be solved exists in the environment of the system that should solve it (it would be embarrassing if it would reside *in* the system) [5]. Figure 2 summarizes the TRADE approach. The *environment model* is an outcome of problem analysis. It represents the environment as consisting of entities, their behavior and their communication. We assume the problem to be solved has been clearly stated and analyzed. The *external requirements* are desired external system properties and include desired system functions, behavior and communication with external entities. The *essential architecture* of a software system is the architecture it would have if we had ideal implementation technology. It is motivated by the structure of the external environment and the system requirements and not by any implementation considerations [8]. Other terms that have been used for this are *logical* and *conceptual* architecture. The *implementation platform* is the collection of programmable entities that will contain the software. These too have functions, behavior and communication. The *implementation architecture* is the mapping of the essential architecture to the programmable entities in the available implementation platform.

3.5 Design Techniques

There are very few techniques needed to specify the aspects listed in Fig. 1. The second column of table 3 lists a few simple techniques that are sufficient.

	Simple	Complex
Decomposition	ERD	SSD
Functions	Mission statement, refined to function descriptions	
Behavior	Event list, possibly including state transition diagram	Statechart
Communication	Communication diagram	

Fig. 3. The techniques in TRADE.

In the following explanation, the term "entity" is used to indicate whatever a description technique is applied to: the entities in the environment, the system itself, or parts of its architecture or of its implementation platform.

– An *entity-relationship diagram* (ERD) represents entity types and their cardinality properties. We restrict the meaning of ERDs to classification and identification (counting) properties of entities. ERDs can be used to represent the decomposition of the environment or of an architecture into entities. It is often used to represent a decomposition of the subject domain into entities. A complex extension of ERDs is the UML *static structure diagram* (SSD), which allows the declaration of services (interfaces, operations, signal receptions etc.) offered by entities, which are now called "objects" in object-oriented methods. Usually there is too much interface detail in a system to be all represented in diagram form.
– *Mission statement* and *function descriptions* describe the things that an entity will do for its environment in natural language. They should emphasize the value delivered for the environment. They can be used to specify the mission and functions of the system and of the entities in the system's architecture.
– An *event list* of an entity is a list of all events that the entity should respond to, and the desired response of the entity. It can be a list of informal natural language descriptions, but this can be refined into state transition tables or diagrams. Statecharts are complex state transition diagrams that can represent information in an event list in diagram form.
– A *communication diagram* consists of boxes and arrows that represent entities and their communication channels. The boxes may represent individual entities or entity types. In the second case, the arrows represent communication channels between instances of the types. A communication diagram may be used to represent communication in the environment, between the system and its environment, between entities in the system architecture, and between entities in the implementation platform.

The appendix contains examples of the use of these techniques, discusses their meaning informally and indicates their use.

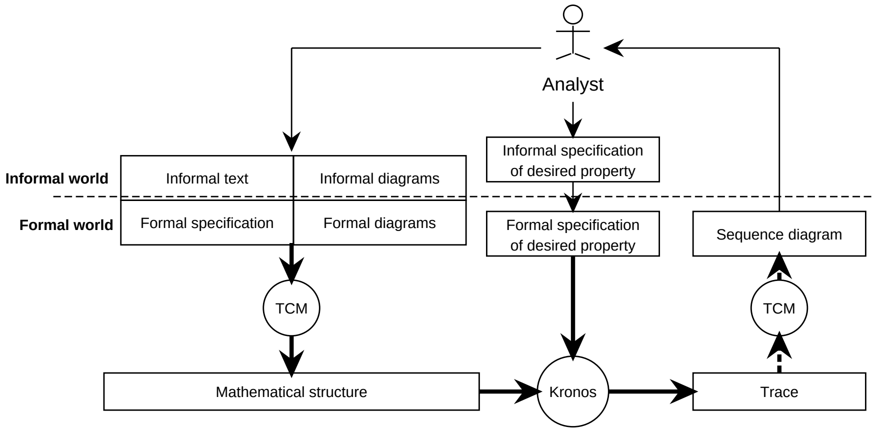

Fig. 4. Combining informal techniques with model checking.

3.6 Formal Techniques

In our view the relationship between the formal and informal parts of a specification (Fig. 4) is that the formal part rephrases fragments of the informal part. The formal part can consist of text or diagrams with a formal semantics. To illustrate the viability of this approach, we have defined a formal execution semantics for object-oriented statecharts that is suitable to represent essential-level behavior [3]. Barring unrestricted object creation, the semantics of a collection of object-oriented statecharts is a finite-state labelled transition system (LTS), which is the mathematical structure in Fig. 4. We have also defined an extension of computation tree logic with actions and real time (ATCTL) to be used as a property language for reactive systems, and defined a translation of ATCTL into the property language of the Kronos model checker [1]. We implemented these using the diagram editing tool TCM [2] as a front-end to Kronos. Space restrictions prevent us from giving more details about this.

The fat arrows in Fig. 4 represent manipulations done by machine. The solid fat arrows have been implemented in TCM, and we used this on our example specification in the appendix to check some desirable properties. The analyst does not have to know or understand the translations behind the fat arrows. The combination of TCM and Kronos thus helps the analyst to understand the design in an early stage without implementing it and without having to learn a complex formal language. It also helps making the informal parts of the model precise.

4 Summary and Further Work

In this paper we have only discussed techniques and showed their place in an approach to reactive system specification and design. We have not discussed precise or formal semantics, or guidelines for using these techniques. A compendium of these guidelines has been prepared elsewhere [14]. Current work includes implementing the dashed fat arrows in Fig. 4 and applying the resulting tool to a number of case studies.

Acknowledgement. The paper benefited from many discussions with and detailed comments of Michael Jackson, and from comments by Jaap-Henk Hoepman on security mechanisms.

References

1. M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: a model-checking tool for real-time systems. In A.J. Hu and M.J. Vardi, editors, *Computer aided verification (CAV)*, pages 546–550. Springer, 1998. LNCS 1427.
2. F. Dehne and R.J. Wieringa. Toolkit for Conceptual Modeling (TCM): User's Guide. Technical Report IR-401, Faculty of Mathematics and Computer Science, *Vrije Universiteit*, De Boelelaan 1081a, 1081 HV Amsterdam, 1996. http://www.cs.utwente.nl/~tcm.
3. H. Eshuis and R. Wieringa. Requirements level semantics for UML statecharts. In S.F. Smith and C.L. Talcott, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS) IV*, pages 121–140, 2000.
4. D. Harel and A. Pnueli. On the development of reactive systems. In K. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498. Springer, 1985. NATO ASI Series.
5. M.A. Jackson. *Problem Frames: Analysing and Structuring Software Development Problems.* Addison-Wesley, 2000.
6. D.N. Jansen and R.J. Wieringa. Extending CTL with actions and real time. In *International Conference on Temporal Logic 2000*, pages 105–114, October 4–7 2000.
7. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent System Specification.* Springer, 1992.
8. S. M. McMenamin and J. F. Palmer. *Essential Systems Analysis.* Yourdon Press/Prentice Hall, 1984.
9. W. Reif and K. Stenzel. Formal methods for the secure application of java smartcards. Technical report, University of Ulm, December 1999. Presentation slides. http://www.informatik.uni-augsburg.de/swt/fmg/projects/javacard_presentatio n.ps.gz.
10. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1999.
11. K. Shumate and M. Keller. *Software Specification and Design: A Disciplined Approach for Real-Time Systems.* Wiley, 1992.
12. UML Revision Task Force. *OMG UML Specification.* Object Management Group, march 1999. http://uml.shl.com.
13. R.J. Wieringa. A survey of structured and object-oriented software specification methods and techniques. *ACM Computing Surveys*, 30(4):459–527, December 1998.
14. R.J. Wieringa. *Design Methods for Reactive Systems: Yourdon, Statemate and the UML.* Morgan Kaufmann, To be published.
15. E. Yourdon. *Modern Structured Analysis.* Prentice-Hall, 1989.

- **Name of the system:** Electronic Ticket System (ETS).
- **Purpose of the system:** Provide capability to buy and use tickets for a railway company using a Personal Digital Assistant and a smart card.
- **Responsibilities of the system:**
  - Sell a ticket
  - Show a ticket on request
  - Stamp a ticket for use
  - Refund a ticket
- **Exclusions:**
  - The system will not provide travel planning facilities.
  - Only tickets for making a trip by one person will be considered.

Fig. 5. Mission of the ETS.

The Electronic Ticket System (ETS)

The Electronic Ticket System (ETS) is a software system by which travelers can buy a railway ticket with a smart card when put in their Personal Digital Assistant (PDA). The ticket is a digital entity than can be created on the smart card itself. Payment is done through a wireless connection with the computer system of the railway company, that itself is connected to the computer systems of a clearing house [9].

External Requirements: Functions and Other Properties

The *mission statement* of a system states the purpose of the system and should be written down for all reactive systems. It lists its major responsibilities and, possibly, things that the system will not do. Responsibilities are things the system should do, exclusions are things the system will not do. There are infinitely many things the system will not do, but writing down a few of these is an important tool in expectation management. Figure 5 shows the mission statement of the ETS.

Functions can be presented in the form of an indented list called a *function refinement tree*. This is useful for all reactive systems, and can be used to bound the discussion about desired functionality with the stakeholders. The current system is so simple that all its functions have already been listed in the mission statement, and a separate function refinement tree is not needed. Each function should be specified from the standpoint of the system, emphasizing the value delivered to the environment. See Fig. 6 for an example.

One of the required ETS properties is that it should offer the functionality described above. Other required properties include the ones listed in Fig. 7. We discuss the formal specification and model checking of these properties at the end of the appendix.

- **Name:** Sell a ticket.
- **Triggering event:** Traveler requests to buy a ticket.
- **Delivered value:** To sell a railway ticket to a traveler at any time and place chosen by the traveler.

Fig. 6. Description of an ETS function.

P1 A traveler cannot get a ticket without paying for it.
P2 A traveler who has paid for a ticket gets it.
P3 A refunded ticket cannot be used any more.
P4 A fully used ticket cannot be refunded.
P5 It is not possible to use a ticket twice.

Fig. 7. Other required ETS properties.

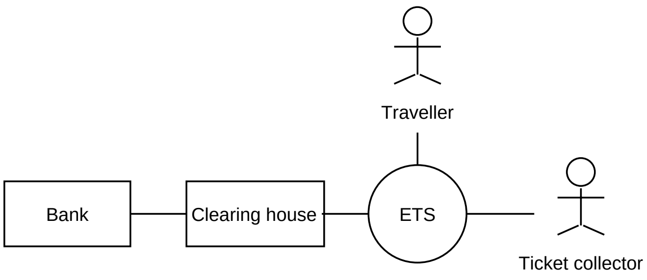

Fig. 8. Communication context of ETS.

Environment: Entities and Communication

Desired properties, including system functions, are provided by interacting with the environment in stimulus-response behavior. Figure 8 shows the communication architecture of the environment, including the external interfaces of the system. The diagram abstracts from the internal distribution of the software system over physical entities of the implementation platform, and from the way communication channels are realized. That distribution will be part of the architecture specification shown later. The context diagram views the system as a black box offering certain functionality and shows which communication channels with the environment exist. It is always useful to draw a communication diagram of a reactive system. It separates the part of the world to be designed (the system) from the part of the world that is given (the environment).

Figure 9 shows an entity model of the subject domain of the ETS. It should be supplemented by a number of constraints, such as that all segments of a route

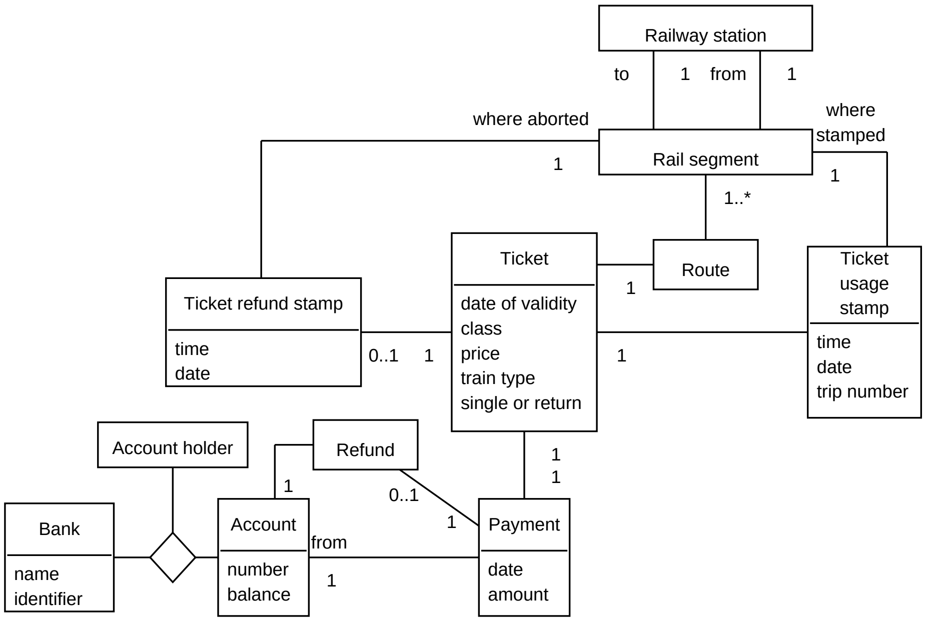

Fig. 9. Subject domain of the ETS.

should be connected. Note that the ERD is a model of the environment, not of the data stored by the system. Note also that the traveler may be different from the account holder.

Whenever there is potential misunderstanding about the meaning of the data that crosses the external system interface, as in data-intensive reactive systems, it is useful to make a subject domain ERD that represents the types of entities to which the external interactions of the system refer.

External Requirements: System Event List

Each function is triggered by an external event, a condition change, or a temporal event. Each function can be refined into a set of possible transactions, where a transaction is an atomic interaction of the system. Each transaction in turn can be represented by a tuple (event, stimulus, current state, next state, response, action), which tells us which stimulus triggers the transaction, which event is supposed to have caused this stimulus, what the response should be, given the current state of the system, and what action is assumed to be caused by the response. The current and next state in the tuple are really states of the dialog between the system and its environment. To perform its function, the system must maintain a representation of this state. In general, there may be a many-many relationship between transactions and functions: One transaction may occur in several functions and one function may contain several transactions.

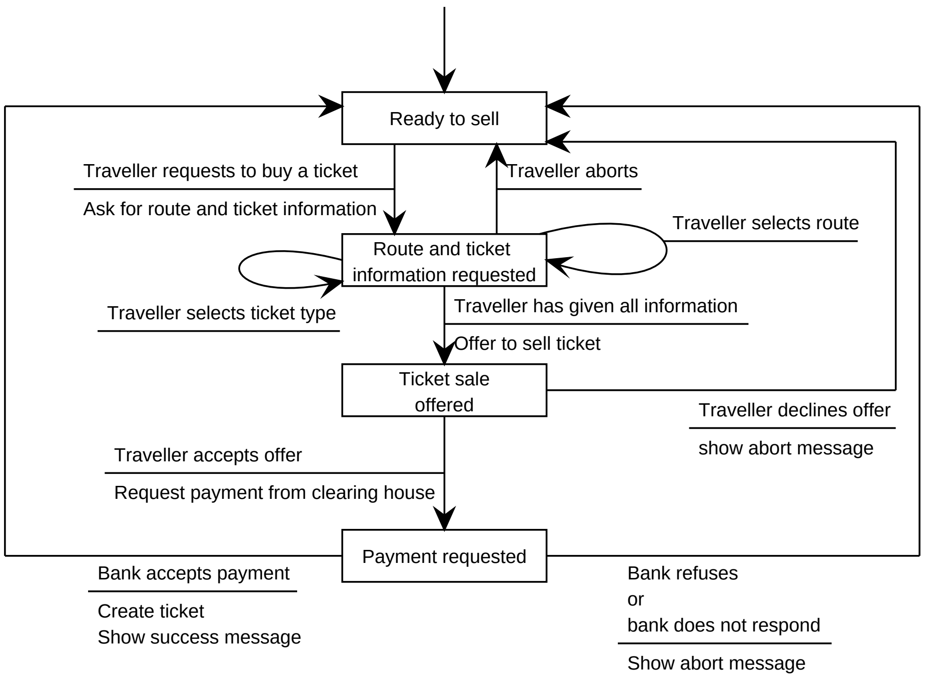

Fig. 10. Selling dialog.

Often, we can represent parts of the system event list by a state transition diagram, e.g. a statechart or its simpler ancestors Mealy and Moore diagrams. Figure 10 gives the event list for the selling function in the form of a Mealy diagram. Rectangles represent states, arrows state transitions, and arrow labels list events above the line and actions below the line.

An event list is always useful to make, but different reactive systems require different levels of detail. Some information systems have merely two types of external events to respond to, namely query and update, but in many other cases there are also state change events and temporal events to respond to. Refining the system function descriptions into an event list uncovers these desired responses.

Architectures and Implementation Platform

Essential Architecture. The essential, distribution-independent architecture of the system is shown in Fig. 11. It uses a hybrid notation in which parallel bars represent data stores and rectangles represent stateful objects.

– The selling dialog (Fig. 10) has been allocated to a single object class, each instance of which can execute this dialog.
– The other functions are simple transformations that produce output or perform updates on request.

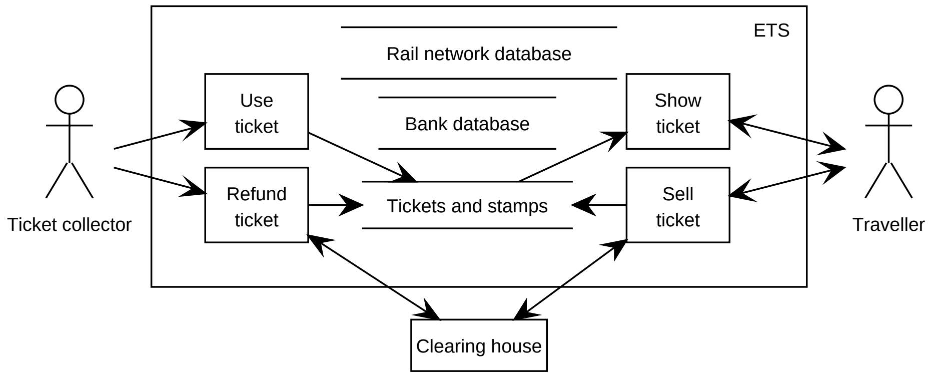

Fig. 11. Essential communication architecture of the system in a hybrid notation

– Data about the subject domain has been partitioned into three data stores.

Figure 12 represents the structure of the data stores. Now we reuse the subject domain ERD to represent the structure of the data and allocate the data to stores. Note that the system does not contain data about all subject domain relationships.

A communication diagram such as 11 is very useful in reactive system design to show simple communication architectures. They can be used to show the high-level communication architecture of a system and simple lower-level architectures, as in the ETS. However, for most systems, at lower aggregation levels they quickly become too complicated to be useful.

Checking Desired Properties. To check whether the essential architecture satisfies all desired properties, we created our model with the TCM editor and generated Kronos input for it. We also formalize the desirable properties in ATCTL, and checked whether they hold in our essential architecture, using Kronos. As an illustration, consider these two properties:

P1 You cannot get a valid ticket without paying. $\neg\exists(\top\ _{\mathbf{-bank\_pay\_yes}}\mathcal{U}\mathsf{valid})$.
P3 A refunded ticket will never become valid again. $\mathsf{refunded} \rightarrow \neg\exists\Diamond\mathsf{valid}$

P1 was shown to be true using Kronos. The negation of P3 was checked with Kronos and proved to be unreachable. Because the negation is unreachable, the property holds.

The meaning of these proofs is that the system, if implemented this way, will be secure. It does not mean that the environment will be secure. For example, it does not mean that an insecure connection to the railway company or the clearing house is impossible. And with such an insecure connection, third parties could masquerade as bank, and cause the railway company to give a ticket to the passenger without valid payment. If that should be excluded, then the system boundary should be extended to include the connections and we should formulate properties of these connections.

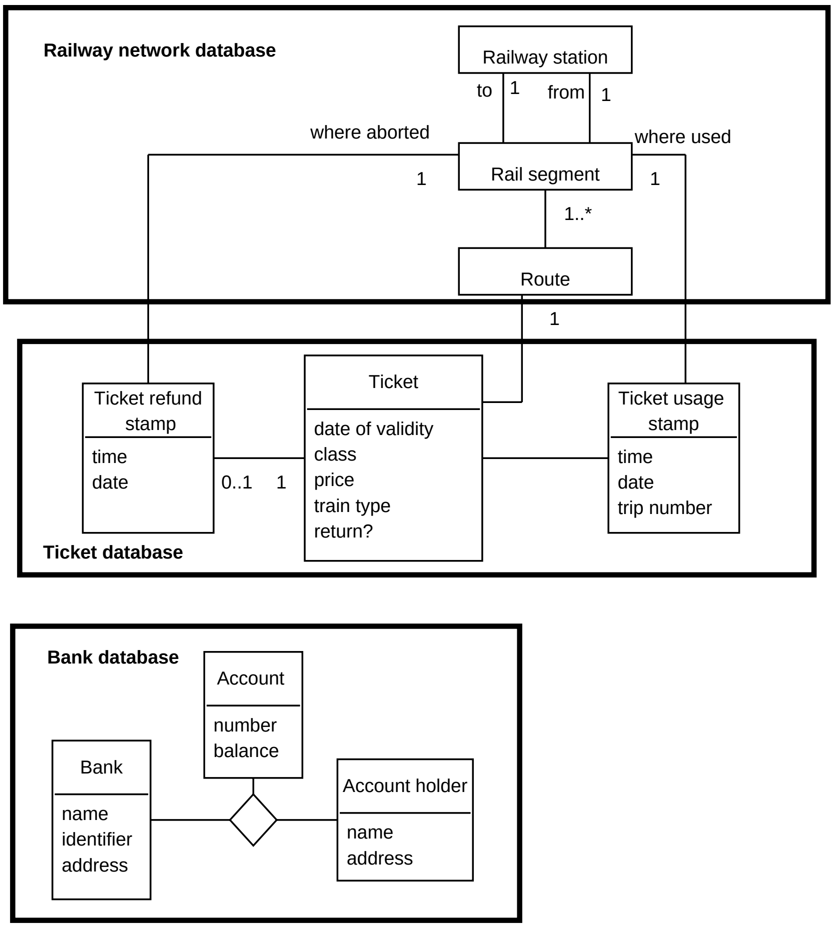

Fig. 12. Data about the subject domain.

Implementation Architecture. The physical distribution architecture of the system is represented by the UML deployment diagram of Fig. 13. This is the implementation platform given to the designer.

Figure 14 shows the allocation of the essential architecture elements of Fig. 11 to the nodes in the deployment network. The following design decisions have been made:

– Functionality is allocated to nodes where it is needed.
– Duplication of data stores is added to avoid frequent communications. The price to pay is that duplicate data stores may be mutually inconsistent. Because the rail network is not likely to change very frequently, and changes are planned far in advance, there is a small risk that this will ever happen.
– Communication interfaces are added.

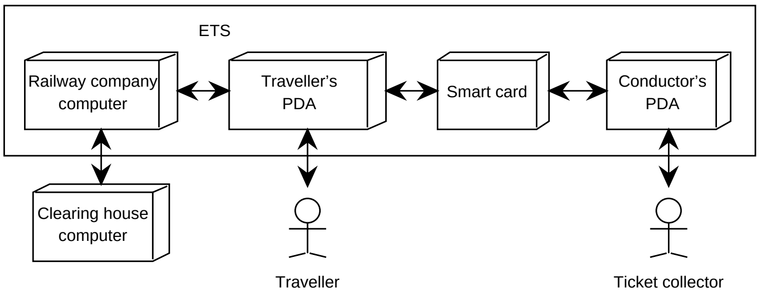

Fig. 13. Physical implementation platform and context.

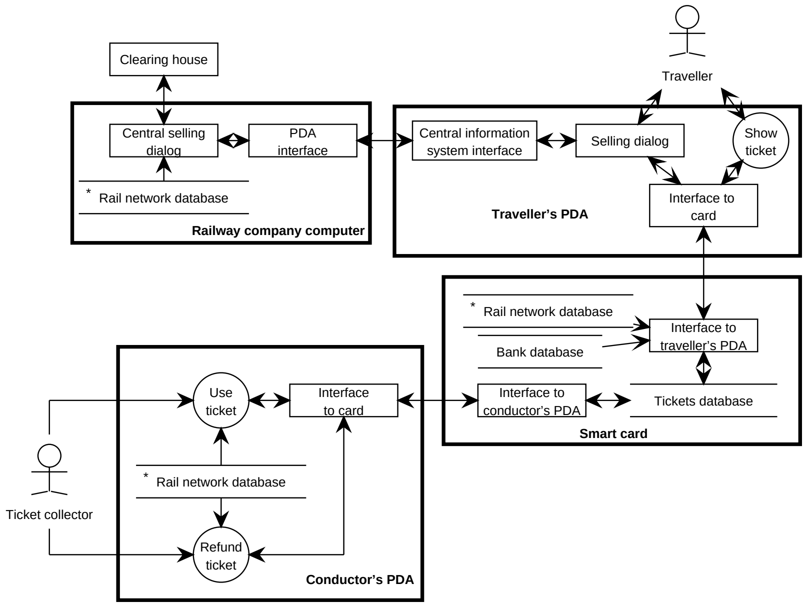

Fig. 14. Allocation of components of the essential architecture to the physical deployment network.

The next stage in creating a secure design would be to check that the implementation is correct with respect to the higher-level design. The implementation is represented by Fig. 14, the data model of Fig. 12 and the textual specifications of all elements of these diagrams (these are not given here). The specification is given by the high-level diagram of Fig. 11 and the textual specifications of the data stores, transformations and object classes contained in it. Both models have a formal semantics in terms of labeled transition systems, which makes classical bisimulation equivalence checking techniques applicable, at least in principle. Working this out for practical examples is subject of future research.