Documenting Design Refinement

E.-R. Olderog
Computer Science Department
University of Oldenburg
D-26131 Oldenburg, Germany
olderog@informatik.uni-oldenburg.de

A.P. Ravn
Computer Science Department
Aalborg University
DK-9220 Aalborg East, Denmark
apr@cs.auc.dk

ABSTRACT

We show how UML class diagrams can be used to document design by refinement in the early design stages. This is illustrated by an example from the area of embedded real-time and hybrid systems. A precise semantics is given for the UML class diagrams by translation to the Z schema calculus.

1. INTRODUCTION

When we wish to introduce formal methods into a software development process, we face the problem of finding the right place for the specification and verification documents within the framework defined by the process. In the following we outline the documents of a typical software development process, the typical steps in a refinement based formal method, and conclude with a possible match of the documents of a formal method to the software process. As already identified in the project KORSO (Correct Software) [23], formal methods may need some top level diagrams. Whereas in [23] the concept of a *development graph* was used for this purpose, we propose to employ UML class diagrams with precise guidelines for the kind of association and dependency relations used.

1.1 Software processes and documentation

A typical state-of-the art process is the V-model [2] which organizes the design and test documents in parallel levels from requirements to code and back again, see Figure 1. Design produces design and test specification documents for each level. The design specifications are input to the lower level, while the test specification is input to the test leg of the model. The outcome of tests are documented in test reports. In the following we focus on documents from:

- the System (Sub-system and Segment) level, where documents describe the organization, architecture or structure of the systems or its major constituents, and
- the Component level with specifications for the smallest individually designed units.

We shall not be concerned with source code documents at the lowest level of Module and Data. Furthermore, we focus on design and test documents and not associated management reports.

In a state-of-the-art organization with a defined and repeatable process, we find the following typical documents:

System Decomposition. The central architectural document that gives an overview of all components and their relations. This document is typically a diagram of some sort (activity diagram, process diagram, context diagram), possibly with subdiagrams and textual descriptions of the main functionalities and interfaces of the constituents.

Component Specification. This defines the protocols and methods and desired behaviour for a given component.

Component Test Specification. A description of the tests to be executed to ensure that the implementation conforms to the (component) specification.

Component Test Report. A report on the successful execution of the specified component tests.

The documents above are key products of the software development process, and a major part of the work of the software engineers or programmers is to produce, update and digest these documents. While work with the specification documents is somewhat creative, the work with test specifications and test reports is more analytical and often very tedious; thus any techniques that may improve the intellectual content or automate the production of these documents are very interesting to a software organization. This is probably the main reason for software organizations to take some interest in formal methods.

1.2 Formal methods

The basis of formal methods is to formulate specifications for components in a notation such that mathematical techniques can be used to derive their properties. A component specification will describe an interface in terms of an alphabet of admissible events or actions together with some

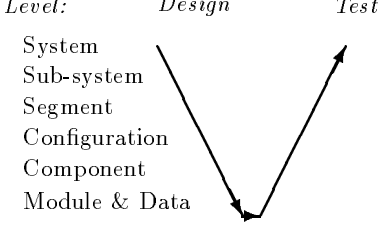


Figure 1: The V-model

description of the language over this alphabet that is the desired protocol. A component specification will describe further the desired behaviour, possibly operationally by a state machine that conforms to the interface protocol or declaratively by a formula in a suitable logic.

Such specifications are not too different from what is usually found in corresponding documents of conventional processes. The difference may even be narrowed further when modern tools that use Message Sequence Charts, State Charts or other State Machine formalisms are taken up by the software producing organizations. The major difference is, when it is tested that the implementation conforms to the specification. Conventional techniques postpone test until a concrete implementation is produced for this level of the design and thus by all underlying levels.

The key to a formal methods approach is that components (specifications or descriptions when we are speaking of implemented ones) may be combined using assorted combinators and related by some "implements" or "refines" relation. The test of conformance is then a proof that such a relation holds for the specific components of a design. Specifications can thus be tested before implementations are available.

When such refinements are done individually for different parts of the system, it is important to document them as part of the system decomposition, just like the test specification and test report together document that testing has been done.

1.3 Formal methods documents

For formal methods we thus expect to see the following *changes* in the documentation.

System Decomposition. This is a structural document that documents not only final components as now, but also intermediate abstractions that are used to formulate refinements.

Component Specification. Apart from notation essentially unchanged.

Component Test Specification. This is replaced by a *Component Refinement Specification.*

Component Test Report. This is replaced by a *Component Refinement Verification* that documents a proof that the refinement holds. This is where automated tools like verifiers and model checkers are extremely important.

The system decomposition document is the topic for this paper. We want to investigate how class diagrams are used to document a refinement based development.

1.4 Overview

We start in Section 2 from a well researched formal approach to real-time and hybrid systems based on the Duration Calculus [34], an interval-based real-time logic and calculus with continuous time domain. It has been demonstrated in the research project ProCoS (Provably Correct Systems) [14] and subsequent work that this logic is suitable for expressing both high level safety and functional requirements for real-time systems as well as the real-time semantics of design specifications and real-time programming languages [22]. An advantage of such a logic-based approach to the description of real-time systems is that the central notion of *refinement* between different design stages can conceptually be modelled by logical implication.

On the other hand, Duration Calculus is just a logic, it lacks a structuring mechanism that is needed in larger applications. A first attempt to overcome this shortcoming is to use the schema calculus of Z [29, 32]. We have done this in previous work, e.g. in [22]. However, with Z we obtain long sequences of schemas that are difficult to follow. That is why we looked at the Unified Modelling Language (UML), a newly emerging standard for the graphic, object-oriented modelling of software-intensive systems [1]. UML embraces various sorts of diagrams for describing both the static and dynamic aspects of systems. Static aspects are covered by class diagrams with inheritance and other relations, and dynamic aspects are covered by state charts, (message) sequence diagrams and other sorts of diagrams. In Section 3 we demonstrate that a small UML subset comprising the concepts of *class diagram, interface, inheritance, aggregation* and *package* suffices to add and visualize the missing structure to design stages formulated in a refinement oriented formal approach. The special UML *realisation* dependency is in Section 4 used to document refinement of components.

The case study of the Generalized Railroad Crossing [15, 22] serves in Section 5 to illustrate our approach.

UML is claimed to be fit for various application domains, even real-time systems [28, 27]. However, such systems critically depend on the semantics of quantitative time constructs, and semantics is one of the weak spots of UML. While there is an informal explanation given for each of the constructs in UML, a consistent formal semantics of the full language is missing. Thus a connection to formal methods with their analysis and verification techniques is not yet established. Currently, various researchers attack the challenge of UML by giving formal semantics for various parts of the language (see e.g. [10]). It is indeed questionable whether a formal semantics for the full language UML is achievable and desirable. We think that it is more appropriate to tailor UML profiles for specific application domains. For such profiles a clear semantics is easier to achieve.

In Section 6 we give a formal semantics for the UML subset used here by translating the graphic notation back to the Z schema calculus. In this process we have taken some

care to be compatible with the informal semantics expressed by the originators of UML for instance in the recent draft standards.

Finally, Section 7 summarizes the paper and discusses related work and further perspectives.

2. A FORMAL METHOD

The application domain considered in this paper are real-time and hybrid systems. These systems consist of some physical *process* that is continuously evolving over time for which a suitable discrete *controller* in software and/or hardware has to be constructed such that the controlled process exhibits a desired time dependent behaviour [12, 30]. The adjective *hybrid* stresses that the process involves real-valued continuous data like temperature, pressure and time. The adjective *real-time* stresses that we are concerned with a subclass of hybrid systems where time is the only real-valued quantity. Real-time systems can also be seen as reactive systems where reactions to certain inputs have to occur within given time intervals [19, 24]. In both cases the interaction between process and controller proceeds via *sensors* and *actuators*. When constructing the controller the reaction times of all components of this system have to be taken into account.

Following [14] a real-time or hybrid system is described by a set of time dependent observables *obs* which are functions

$$obs : Time \to D_{obs}$$

where *Time* is a continuous time domain, the non-negative real numbers, and D_{obs} is the data domain of *obs*. For real-time systems D_{obs} will be a discrete or even finite set, for hybrid systems D_{obs} will be a subset of the real numbers.

Properties $P(obs)$ of observables *obs* can be expressed in many different notations, e.g. logical formulae, behavioural diagrams or state-transition models. Since we want to state and verify system properties, we request that these notations have a formal semantics. We found Duration Calculus (DC for short) [34], an interval-based real-time logic for specifying and proving such properties, a convenient unifying framework for this purpose. DC can be used both as an explicit specification logic and as a semantic basis for other specification techniques for real-time and hybrid systems. The papers [7, 4, 6, 26, 25] show how the semantics of graphic notations for expressing requirements and designs can be given in terms of DC. However, DC is not the topic of the paper, which can be read with any logic based notation in mind.

2.1 Refinement

In a logic based framework, different syntactic descriptions of a real-time system are given by predicates in the same logic. The advantage is that then the central concept of *refinement* between different design stages corresponds to logical implication between predicates.

For example, if *req* is a requirement and *spec* a design specification we write

$$spec \Rightarrow req$$

to express that the semantic predicate associated with *spec* logically implies the semantic predicate associated with *req*. Thus the set of behaviours allowed by *spec* is contained in the set of behaviours allowed by *req*. Conceptually, this means that *spec refines* or *satisfies* all the properties of *req*. In other words, *spec* is *correct* w.r.t. *req*.

Analogously, if *spec* is a specification and *prog* a program then

$$prog \Rightarrow spec$$

expresses that *prog* correctly implements *spec*. By the transitivity of $\Rightarrow$ we can deduce $prog \Rightarrow req$ from the above two facts, i.e. *prog* correctly implements the requirement *req*.

This picture gets more complicated if the predicates associated with, say, *prog* and *spec* involve observables on different data domains, say *prog* involves more concrete observables *c* and *spec* more abstract ones *a*. Then we need a *linking invariant* that relates the values of *a* and *c*. Such an invariant is known from data refinement [16, 3]; it can also be expressed as a predicate, say *link*. Then refinement can be expressed by the formula

$$prog \wedge link \Rightarrow spec, \qquad (1)$$

i.e. the conjunction of (the semantic predicate of) *prog* and *link* together imply (the semantic predicate of) *spec*. This treatment of refinement follows the philosophy that "programs are predicates" [17].

3. DOCUMENTING COMPONENTS

A system component consists of a declaration and a property describing the allowed behaviour of the declared entities. Structuring mechanisms are needed to build larger components out of simpler ones. It is well known that in software development one needs suitable means for documenting components and their structure (see e.g. [23]).

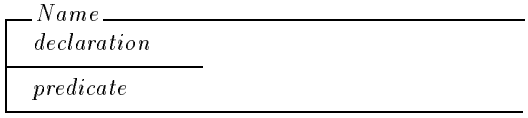
3.1 Z schemas

An approach from the formal methods area is the specification language Z, which is particularly suited for the description of complex state spaces and their transformation [29, 32, 33]. Z is based on set theory and predicate logic and adds structure to it by the concept of a *schema* and the *schema calculus*.

A schema groups together a declaration part and a predicate part under a schema name. This is written as

$$Name \mathrel{\widehat{=}} [\, declaration \mid predicate \,]$$

or as a schema box



The declaration introduces logical variables ranging over certain data domains and the predicate describes properties of these variables. For a schema *Name* let $Decl_{Name}$ denote its declaration part and $Pred_{Name}$ its predicate part.

In the application domain of real-time and hybrid systems we consider schemas S of the form

$$\begin{array}{|l} S \\ \hline obs : Time \to D_{obs} \\ \hline P(obs) \\ \hline \end{array}$$

where $P(obs)$ as before is a predicate specifying properties of obs.

To document the structure of more complex classes one can use the Z schema calculus. First of all, it allows us to *include* schemas in other schemas by referring to their name. For example,

$$\begin{array}{|l} T \\ \hline S \\ x : D_x \\ \hline Q(x) \\ \hline \end{array}$$

denotes the schema

$$\begin{array}{|l} T \\ \hline obs : Time \to D_{obs} \\ x : D_x \\ \hline P(obs) \\ Q(x) \\ \hline \end{array}$$

where the list of predicates $P(obs)$ and $Q(x)$ stands for their conjunction $P(obs) \wedge Q(x)$.

Schemas can be combined by logical operations like disjunction and conjunction. Schema conjunction

$$S_1 \wedge S_2$$

yields a schema U obtained by double schema inclusion

$$\begin{array}{|l} U \\ \hline S_1 \\ S_2 \\ \hline \end{array}$$

In other words, both the declaration parts and the predicate parts of S_1 and S_2 are conjoined.

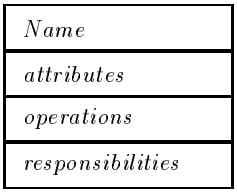
Z is equipped with a simple type system for all its expressions. For a given expression, the Z type approximates the data domain from which this expression can take its values. In particular, schemas themselves are allowed as types. Such a *schema type* corresponds to a record type. For example, the declaration

$$v : T$$

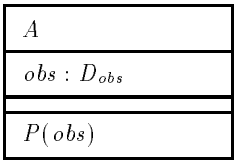
specifies that v is a variable ranging over records with two components, viz. $v.obs : Time \to D_{obs}$ and $v.x : D_x$.

3.2 UML classes

In object-oriented design, a component is described by a class. A *class* has a name and is in essence a state machine where the state is defined by the *attributes* of the class and the events are defined by the *operations* of the class. It is important to note that the state of a class can evolve over time. Restrictions on the state behaviour can be defined as *responsibilities* (for implementing the class). In UML [1] a class is represented graphically as follows:



In the application domain of real-time and hybrid systems we consider classes A of the form



where the attributes describe observables obs that evolve over time and take values in the domain D_{obs}. Note that in contrast to the Z schema S, the time dependency of obs is left implicit here. The operation field is empty because we pursue here a purely state-based approach where a system is described by observables. Operations or methods would correspond to an explicit notion of events. These might be included in an extension of our approach.

In the application domain of real-time and hybrid systems the attributes can be classified into *input, output, controllable* and *parameter*. In UML this classification can be modelled by adding stereotypes such as $\ll input \gg$ to the list of attributes. Attributes classified as $\ll parameter \gg$ are treated as "frozen" in the sense of UML. This signals that these attributes are not dependent on time.

To document the structural relationship between classes one can use UML diagrams. Out of simple classes more complex ones can be built up incrementally using the concept of *generalization* or *inheritance*. In UML this structural relationship is visualized by an inheritance arrow:

$$A \lhd\!\!- B$$

expresses that class A generalizes class B or vice versa that B extends A by inheriting all the attributes and responsibilities of A. It is also possible that a class B extends several other classes $A1, ..., An$. This is visualized as a multiple inheritance (see Fig. 2).

Aggregation is another structural relationship expressing that given classes, say $A1, ..., An$, are part of a larger class B. This is visualized as shown in Fig. 2.

A more general structural relationship is the *association*. It can stand for any relation between classes A and B and is

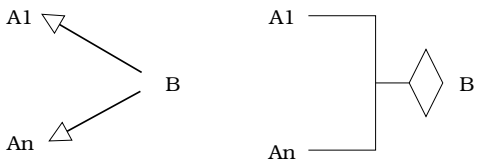


Figure 2: Multiple inheritance and Aggregation

visualized as follows:

$$A \overset{Ass}{\text{———}} B$$

Here the name Ass hints at the intended relationship.

Several classes and relationships can be grouped together into *packages*. These can be referred to by the names that can be stated on their tags. *Notes* are comments that can annotate any UML element, be it a class, a package, a relationship or even an individual attribute. The corresponding UML symbols are shown in Fig. 3.

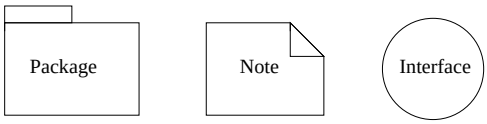


Figure 3: Package, Note and Interface

In UML an *interface* describes a signature, i.e. a collection of operations. This is too restrictive for our purposes, first because we consider only attributes and second because we want to include properties (responsibilities) of the attributes as well. Thus we generalize this concept and allow any class to play also the role of an interface. This role is visualized by the stereotype $\ll interface \gg$ in the name field of the class. Within a package an interface can play the role of an *assumption*, i.e. of a class that we assume to be given in a further construction of new classes, or it can play the role of a *commitment*, i.e. of a class that represents the result of a construction that can be exported to other packages. If further details are not needed an interface can also be represented by a circle containing the interface name as shown in Fig. 3.

4. DOCUMENTING REFINEMENT

Besides the structural relationships among components we also need to document the semantic relationship of *refinement* between components. In Z this can be documented textually using *schema implication*

$$C \Rightarrow A \tag{2}$$

meaning that for the schemas C and A the following holds:

$$Decl_C \subseteq Decl_A \text{ and } \forall\, Decl_C \bullet Pred_C \Rightarrow Pred_A.$$

If the schemas C and A describe components involving the same observables then (2) documents that C refines A.

If C and A involve observables on different data domains, say A abstract data and C concrete data, a linking invariant between abstract and concrete data is needed. It can be documented by a suitable Z schema $Link$ so that the refinement relation between C and A is schema conjunction and implication as in (1):

$$C \wedge Link \Rightarrow A$$

In UML an explicit diagrammatic concept is available, the semantic relationship of *realization* between classes:

$$C \; - - \triangleright \; A$$

expresses that the class A specifies a contract that class C guarantees to carry out. In other words, C realizes the contract of A. This relationship will be used to document refinement between design stages modelled by C and A provided the same attributes are involved.

In case a linking invariant is needed to cope with different data domains of the attributes in $C(oncrete)$ and $A(bstract)$, the refinement relation between C and A can be documented in UML using the pattern shown in Fig. 4. We consider the

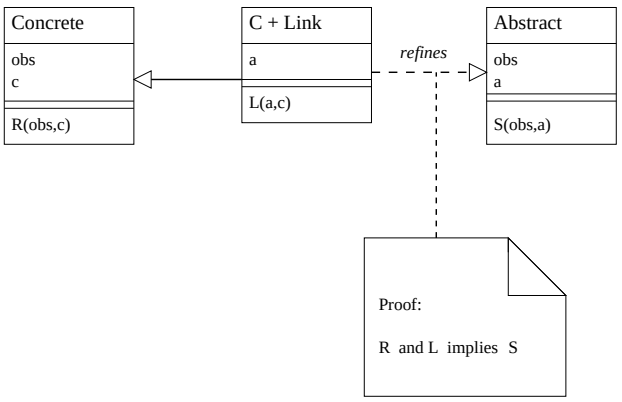


Figure 4: Refinement Pattern with Link

general case where a class `Concrete` has attributes obs and c with a responsibility $R(obs, c)$ and where a class `Abstract` besides obs has another attribute a with a responsibility $S(obs, a)$. To show that `Concrete` refines `Abstract`, a link has to be built between a and c. This is documented by a class `C+Link` with a responsibility $L(a, c)$ extending `Concrete`. Thus `C+Link` has the attributes obs, c, a, and we may use the realization arrow to express the desired refinement relation between `Concrete` and `Abstract`. That this relation really holds has to be proved by verifying the logical implication $R \wedge L \Rightarrow S$ concerning the responsiblities. This proof is documented as a note attached to the realization arrow.

5. CASE STUDY

Let us illustrate the advantages of a combination class diagrams and refinements by the Generalized Railroad Crossing (GRC for short) due to [15]. The informal statement of the problem in that paper is as follows:

> The system to be developed operates a gate at a railroad crossing. The railroad crossing I lies

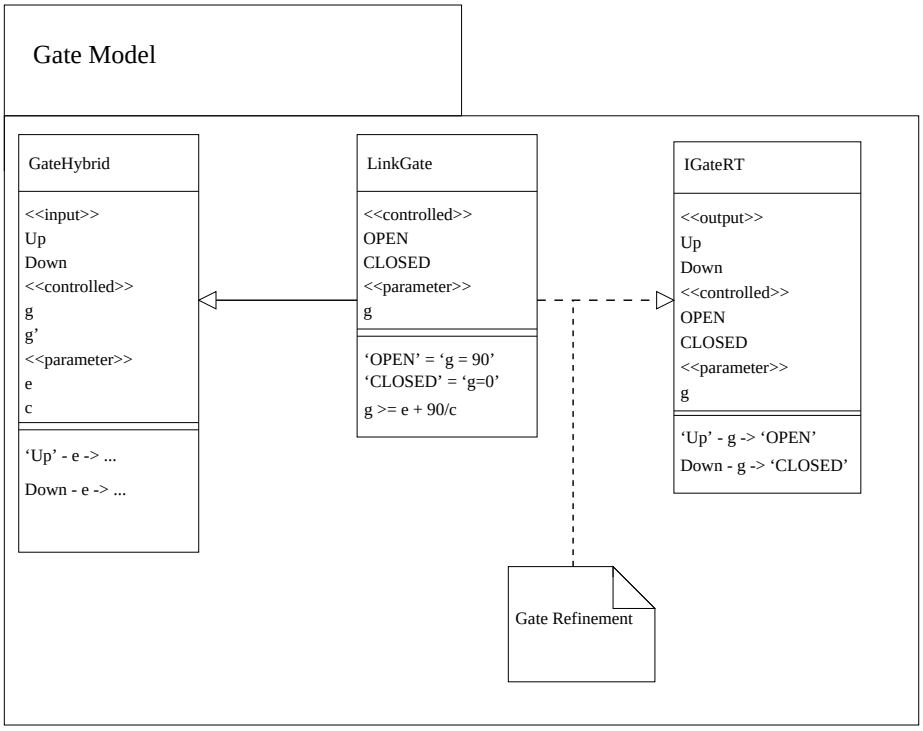


Figure 7: Gate model

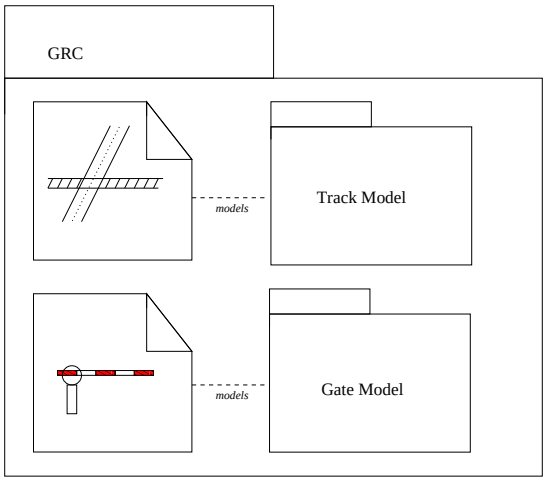


Figure 5: GRC model

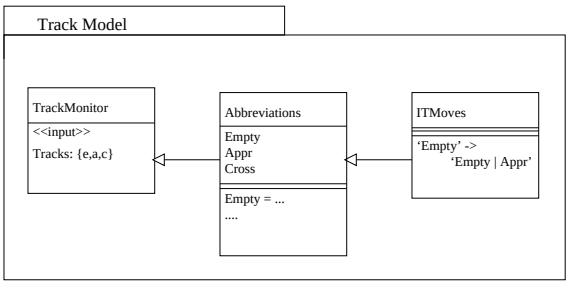


Figure 6: Track model

in a region of interest R, i.e. $I \subseteq R$. A set of trains travels through R on multiple tracks in both directions. A sensor system determines when each train enters and exits region R. To describe the system formally, we define a gate function $g(t) \in [0, 90]$, where $g(t) = 0$ means the gate is down and $g(t) = 90$ means the gate is up. We define a set λ_i of *occupancy intervals*, where each occupancy interval is a time interval during which one or more trains are in I. The ith occupancy interval is represented as $\lambda_i = [\tau_i, \nu_i]$, where τ_i is the time of the ith entry of a train into the crossing when no other train is in the crossing and ν_i is the first time since τ_i that no train is in the crossing (i.e., the train that entered at τ_i has exited as have trains that entered the crossing after τ_i).

Given two constants ξ_1 and ξ_2, the problem is to develop a system to operate the crossing gate that satisfies the following two properties:

Safety Property: $t \in \cup_i \lambda_i \Longrightarrow g(t) = 0$ (The gate is down during all occupancy intervals.)

Utility Property: $t \notin [\tau_i - \xi_1, \nu_i + \xi_2] \Longrightarrow g(t) = 90$ (The gate is up when no train is in the crossing.)

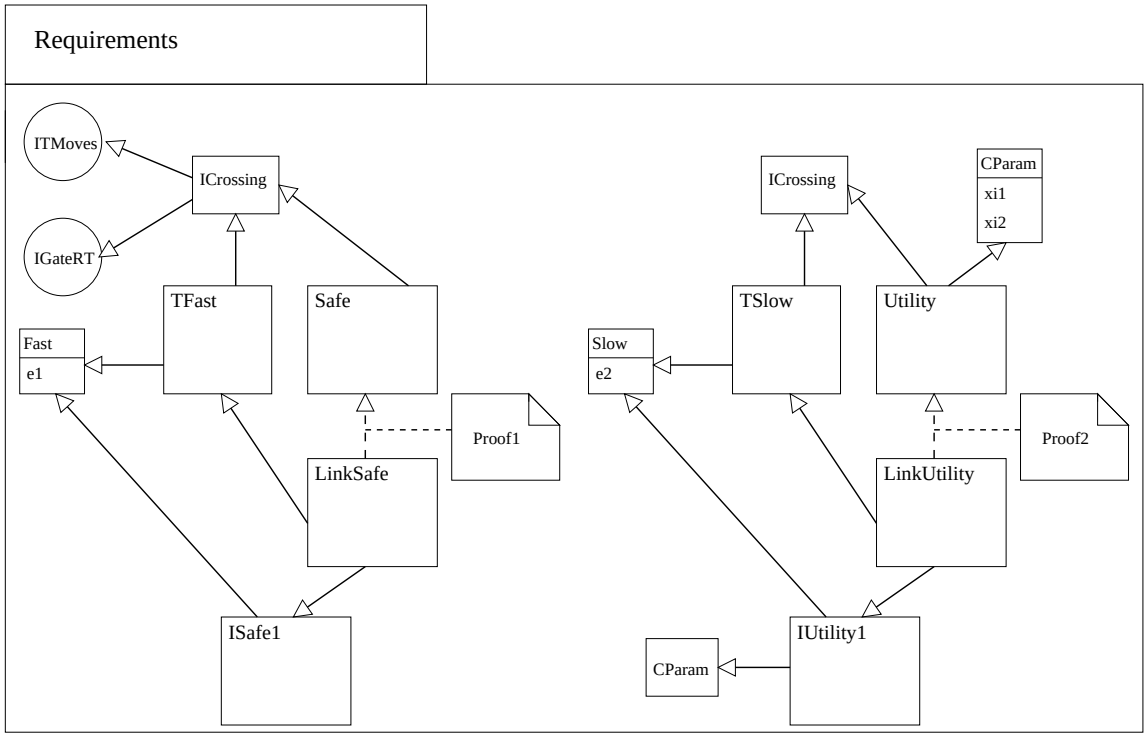


Figure 8: Requirements

The top level view on the UML model of the Generalized Railroad Crossing is given as a package `GRC` containing local packages for the track and the gate model (see Fig. 5). Appropriate notes picture the real world track and gate to remind us of what the local packages are modelling.

First we detail the track model (see Fig. 6). To this end, we introduce appropriate attributes and formalize the responsibilities in DC on the basis of our earlier work [22]. We start with a class `TrackMonitor` where as attribute we take

$$Tracks : \{e, a, c\}.$$

Here e means that the tracks are empty, a means a train is approaching and c means it is in the railroad crossing I. It is convenient to introduce abbreviations, e.g.

$$Empty \text{ abbreviates } Tracks = e, \tag{3}$$

and analogously for $Appr$ and $Cross$. This is done by generalizing the class `TrackMonitor` to a class `Abbreviations` that introduces new attributes $Empty$, $Appr$, $Cross$ and states their relationship (3) to the old ones as responsibilities. Neither the observable $Tracks$ nor the abbreviations $Empty$, $Appr$, $Cross$ document in which order the observable $Tracks$ can assume their values. To align our model with the real world we must formalize the train movement. When a train moves into the region R of interest the track state changes from $Empty$ to $Appr$ and then to $Cross$. After $Cross$ the track state can either change to $Empty$ or to $Appr$ (next train coming). This information is documented in the class `ITMoves` which extends `Abbreviations`. In this case study, the initial letter `I` signals that `ITMoves` is an interface that will be used for further construction in other packages. It is what the remaining system needs to know about trains and tracks.

Next we detail the gate model (see Fig. 7). This is more complex because we start here from a class `GateHybrid` where we model the gate behaviour by continuous observable $g : Time \to [0, 90]$ and its derivative $g^\bullet$ [22]. For the further design of the gate control, however, it is sufficient to work with a more abstract real-time view `IGateRT` of the gate where instead of the continous gate movement we observe only whether the gate is $OPEN$ or $CLOSED$. That this is a correct abstraction of `GateHybrid` or vice versa that `GateHybrid` refines `IGateRT` requires a proof involving a link between g and $OPEN$ and $CLOSED$. This refinement proof follows the pattern shown in Fig. 4. The letter `I` signals again that `IGateRT` is an interface that will be used in further packages.

Once we have built models for the track and the gate we turn to the requirements the gate control should satisfy (see 8). These are the Safety and Utility Property stated above. To formulate these properties we start from the interfaces `ITMoves` and `IGateRT` and collect them in a common interface called `ICrossing`. Now the desired properties can both be formulated as extensions of this interface leading to classes `Safe` and `Utility`. Recall from the problem definition that `Utility` additionally incorporates the parameters (constants) ξ_1 and ξ_2; they have been grouped here in a separate class `CParam`.

In [22] we have shown how to refine the requirements `Safe` and `Utility` into simple controller specifications $Safe1$ and

*Utility*1. The interesting point about these refinement steps is that they succeed only under additional *assumptions* on the environment, here the train speeds. The refinement of `Safe` to a class `ISafe1` assumes that the trains are not too fast. This is modelled by a class `TFast` where the responsibility is given by some DC formula. In UML an assumption can be introduced into the GRC model by extending the class `ICrossing` by `TFast`. Now the refinement of `Safe` to `ISafe1` follows the pattern of Fig. 4 with a suitable `LinkSafe`.

Similarly structured is the refinement step from `Utility` to `IUtility1`. It assumes that the trains are not too slow. This is modelled by a class `TSlow`. The documentation of the two refinements is shown in Fig. 8. Altogether `ISafe1` and `IUtility1` are new interfaces that have been constructed from interface `ICrossing` by incorporating the end user requirements.

The next phase is the controller design documented in Fig. 10. We follow here the approach of [6] where the GRC case study was treated with the aim of producing a control program for a *Programmable Logic Controller* (PLC), a widespread implementation platform in automation technology [18, 21]. In [4, 5] PLC-Automata have been developed as a specification technique for PLC behaviour. PLC-Automata fit well to our formal method because their semantics has been defined in terms of DC.

In [6] it has been shown that the railroad crossing [15] can be controlled by the PLC-Automaton in Fig. 9. It consists

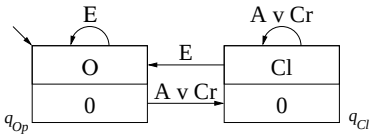


Figure 9: PLC-Automaton

of two states q_{Op}, q_{Cl} and reacts to inputs E, A, Cr by outputting O and Cl. Inputs mark the transitions and outputs appear inside the states. In its initial state q_{Op} (marked by an in-going arc) the automaton outputs O (gate open) and stays in this state as long as the input is E (track empty). Once an input A or Cr (train approaching or crossing) is detected, the automaton switches to state q_{Cl} where the output is Cl (gate closed). As long as the input remains A or Cr, the automaton stays in that state. Once an input E is detected, it switches back to its initial state q_{Op}.

As for physical PLCs, the real-time behaviour of a PLC-Automaton model is determined by an additional parameter, the cycle time ε. It determines how long an input has to be stable before we can be sure that it is read by the PLC-Automaton. Additionally, timing annotations can appear in the lower parts of states. In this simple example we see only the inscription 0. It specifies that the automaton should react as fast as possible to the input values, i.e. within the cycle time.

The details of the real-time behaviour of the PLC-Automaton are defined by its DC semantics [4]. It describes the behaviour of its input, state and output by three observables:

$$in : Time \to \{E, A, Cr\},$$
$$st : Time \to \{q_{Op}, q_{Cl}\},$$
$$out : Time \to \{O, Cl\}.$$

Thus PLC-Automata can be viewed as a graphic notation for expressing properties of the specific observables in, st and out.

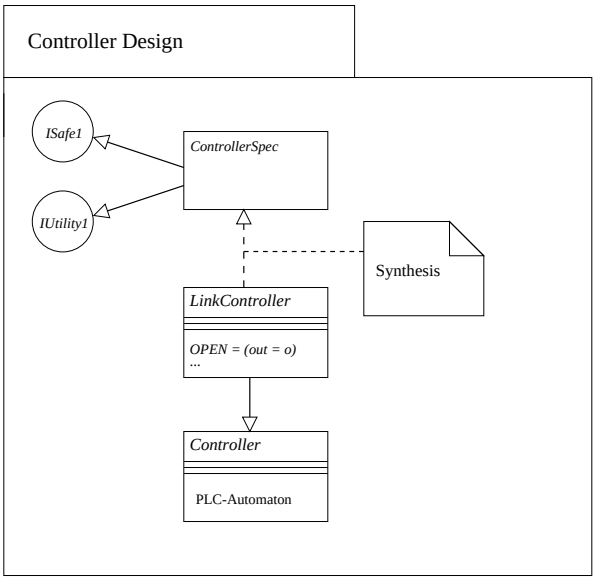


Figure 10: Controller Design

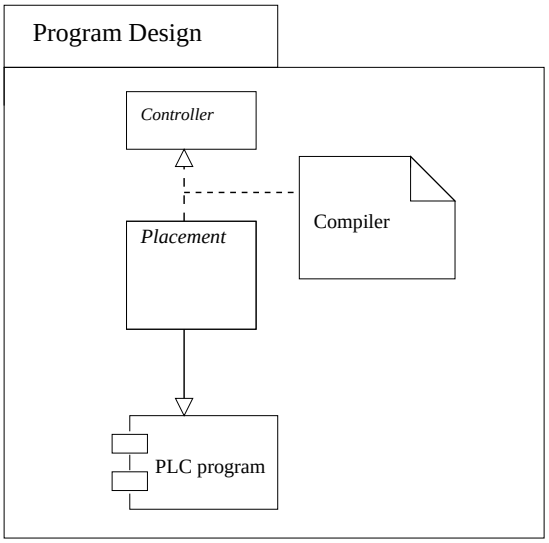


Figure 11: Program Design

Altogether the controller design is just another instance of the general refinement pattern. The design starts from a specification that is obtained by collecting the interfaces `ISafe1` and `IUtility1` in a class `ControllerSpec`. We show that this class is refined by a class `Controller` describing the observables in, st and out and their behaviour by the

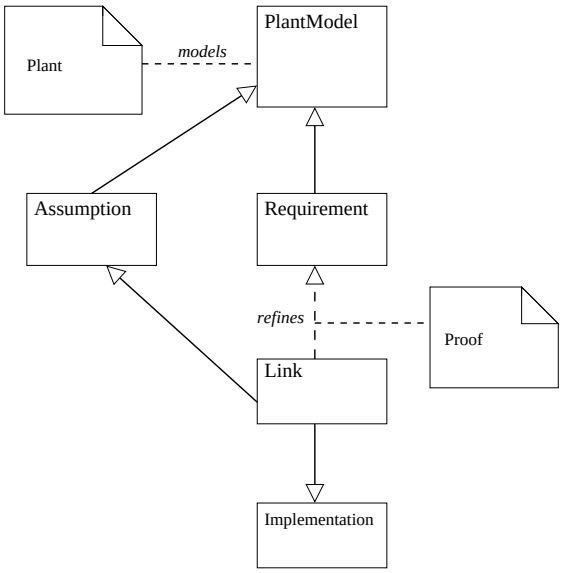


Figure 12: Design Refinement Pattern

PLC-Automaton shown in Fig. 9. Clearly, the refinement relies on a class `LinkController` formalizing the link between the observables of the GRC model, viz. *Empty*, *Appr*, *Cross*, *OPEN*, *CLOSED*, and the observables of the PLC-Automaton, viz. in, st, out. Interesting is that in this case the proof of the refinement can be done fully automatic. In fact, it has been shown in [4] that from specifications of a certain subset of DC, to which the responsibilities of the classes `ISafe1` and `IUtility1` belong, PLC-Automata refining these specifications can be synthesized fully automatically. Thus the refinement proof lies in the correctness of the synthesis algorithm. This is documented in Fig. 10 by a corresponding note.

The last phase is program design. Here the PLC-Automaton as documented in `Controller` is refined to a real PLC program in a suitable programming language like ST (Structured Text) [18, 21]. The link between the abstract specification of the PLC-Automaton and the concrete PLC program is described by the class `Placement`, which extends the component `PLC program`. The refinement relation between `Controller` and `PLC program` is established by a compiler [4]. Fig. 11 documents the program design.

5.1 Design refinement pattern

The overall pattern of a document on design by refinement is shown in Fig. 12. We begin with a `PlantModel` formalizing the real world plant. This model is then extended by formalizing the `Requirement` for the implementing controller software. Showing that this `Implementation` satisfies or refines the `Requirement` asks for a proof that is in general based on a semantic link between different data domains used in the `Requirement` and the `Implementation` and on further assumptions on the plant environment. Using UML class diagrams and relationships this pattern is documented as shown in Fig. 12. As we have seen in the above case study actual designs use several such refinement steps.

6. SEMANTICS

A critique against UML is its missing formal semantics. For the UML subset used here we give such a semantics by translating the graphic notation back to the Z schema calculus. We consider UML class diagrams of the form

$Name$
$attributes$
$responsibilities$

where the responsibilities are given as a predicate. To each UML class of this form we associate a Z schema denoted by $[\![Name]\!]$ as its semantics.

The definition proceeds inductively over the structure of UML inheritance arrows and aggregation relationships.

Induction basis. Consider a class

A
$x : D_x$
$P(x)$

without any outgoing arrow. Then $[\![A]\!]$ denotes the Z schema

$$\begin{array}{|l} A \\ \hline x : Time \to D_x \\ \hline P(x) \\ \hline \end{array}$$

Note that the Z semantics makes explicit the dynamic nature of the class variable x by interpreting it as a time dependent logical variable. Only if in A the attribute x were classified as a $\ll parameter \gg$, it will be interpreted as a "frozen", i.e. time independent variable $x : D_x$ in the Z schema $[\![A]\!]$.

Induction step. Case of inheritance. Suppose the semantics of the class with name A is already defined and a class with name B inherits A, i.e.

$$A \lhd\!\!- B$$

where the class with name B is of the form

B
$y : D_y$
$Q(y)$

Then the semantics $[\![B]\!]$ is given by schema inclusion:

$$\begin{array}{|l} B \\ \hline A \\ y : Time \to D_y \\ \hline Q(y) \\ \hline \end{array}$$

Correspondingly, multiple inheritance means multiple schema inclusion.

Case of aggregation. Suppose the semantics of the classes A and B is already defined and a class with name C of the form

C
$z : D_z$
$R(z)$

aggregates A and B as shown in Fig. 2. Then C is the disjoint union of A and B. In Z this can be modelled introducing two distinct selectors, say a and b, of the schema types of A and B. Using this construction the semantics $[\![C]\!]$ is given as follows:

C
$a : A$ $b : B$ $z : Time \to D_z$
$R(z)$

This semantics of aggregation is as in [9]. This completes the inductive definition.

For classes A and B we define the realization arrow

$$A - - \triangleright B$$

as a semantic relationship denoting the schema implication:

$$[\![A]\!] \Rightarrow [\![B]\!]$$

Associations $A \overset{Ass}{\text{———}} B$ can be easily formalized using a Z schema that makes the association relation Ass between the classes A and B explicit.

7. CONCLUSION

We have discussed how refinement based reasoning in formal methods may be incorporated as an alternative to testing in a conventional software development process. In order to ensure proper documentation of the process, we must provide architecture documents that give the structure of the system in terms of main constituents and their relations. For that purpose we have shown how UML class diagrams are used to describe the structure of refinement based designs. The resulting diagrams have a straightforward semantics that can be expressed by expansion to schemas in the Z-notation. The approach is demonstrated on the well-known case study of the Generalized Railroad Crossing.

This work has grown out of our need to reconcile what we see when we visit good software developers in industry, the object-oriented design methods that most of our students learn, and how we illustrate designs when we introduce examples when we teach or speak about formal methods. From our very first investigations of applications it has been clear that one needs some graphs to illustrate the relation between constituents of a design, which is also supported by the use of architectural diagrams in other applications of formal methods. However, most diagrams are made up to illustrate particular concepts or particular notations. Informal, widely used design methods, including the object-oriented methods have experienced the same multitude of notations, see for instance the excellent survey [31]. This state of affairs explains the enthusiastic reception of the "Unified Modeling Language" (UML). There is a need for a standardisation of the notation for graphs that illustrate the structures of software intensive systems.

UML is a very rich language, because it strives to support different views of a system. There are diagrams for the structural aspects of a *design* as well as the behavioural aspects of the individual reactive *processes*. These are the diagrams that are highly relevant for formal methods. We have focused on the structural aspects in designs because they relate directly to design by refinement. The behavioural aspects describe dynamic properties. In our setting they are given as constraints (UML: responsibilities) in notations that feed into proof systems with which we are familiar. They are used in proofs which are notes in the diagrams. These aspects are orthogonal to the structure of designs, and it would take us too far from our current investigation to contemplate how the UML notations for statechart diagrams and interaction diagrams (a generalisation of message sequence charts) would be given precise semantics such that proofs, including model checking, can be done. Almost outside the scope of formal methods are the UML features for scenarios or *use cases* for clarifying requirements, source and object code components in libraries and files of an *implementation*, and *deployment* of executable modules on processors.

7.1 Related Work

In [23] *development graphs* are proposed for the documentation of a software development process. A development graph consists of nodes representing so-called *units* and of edges representing formal relations between units. A unit can be a software module, a property document or a justification. A relation can be a structural relation, a property relation or a development relation. Examples for development graphs are drawn from the realm of algebraic specifications and functional languages. Conceptually, development graphs play the role of the UML diagrams in our approach. The main difference is that development graphs have a much richer structure than the subset of UML diagrams used here. We find that this structure would have been too rich for the aims pursued here: the documentation of refinements of real-time systems. Our aim was to identify a particular small, yet powerful subset of standard UML for documenting design refinement.

There exist a number of formalisations for subsets of UML [10]. Directly related work is the formalisation of UML class diagrams in Object-Z [20]: their semantics for classes and associations is very similar to ours. Also the RoZ tool [8] shows how a precise semantics [9] supports design documents where UML class diagrams are used in connection with Z schemas. An attempt at integrating also the behavioural aspects is found in the Z-based formalism reported in [11]. However, refinement and the structure of the design process is not part of these works.

It is also noteworthy that many commercial tools that supports behavioural specifications using statecharts or notations like SDL are introducing UML class diagrams to describe the structure of their designs. Assisted by T.K. Jensen we have made a small experiment with the Rational Rose tool which can reproduce the most complex diagram shown here (Figure 8 Refinement). This tool also allows constraints in an arbitrary syntax, so it could be the core of a professional development tool supporting design by refinement. It would need to interface to proof assistants and model checkers through the source texts for the diagrams, cf., the RoZ tool [8]. We noticed that the tool did not complain about interfaces with attributes. If this restriction should be enforced, it would be possible to use visibility constraints on an ordinary class within a package to define an interface – UML is not entirely orthogonal here. Otherwise we have done our very best to use UML features as intended by its developers and explained in the User Guide [1] and the draft standard documents. In particular, we have chosen to use the specific realization arrow for refinement instead of more loose dependency arrows.

7.2 Perspective

The patterns for refinement and design refinement (Figures 4 and 12) are also found in the Gas Burner case study [14] and in the more recent ELPRO railroad control system [5]. The latter example turned out to have a richer data structure describing track topologies and train placements. We did not elaborate these sequential programming facets, but class diagrams would be a possible description technique. More instructive was the use of aggregation to combine descriptions (in this case sensor and actuator pairs) into more manageable objects and association classes to define their placement in the topology.

In order to see how the patterns would fit another approach to verification, we examined the structure of a model checking case study where behavioural properties of an audio protocol were checked with the UPPAAL model checker [13]. Here the `SystemModel` is built as a collection of UPPAAL real-time automata running in parallel. The user requirements are modelled by a class `Properties` extending `SystemModel` and decribing further real-time automata running in parallel to the system automata. The class `OK` represents the success of the verification. It can for example express that certain undesirable states of the automata are not reachable. The UPPAAL tool checks then that the automata of `Properties` refine `OK`. We hope that the reader agrees that the refinement pattern is recognisable in the diagram in Figure 13.

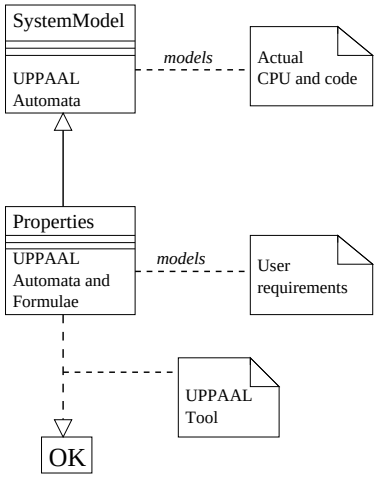


Figure 13: Model Checking as Refinement

Acknowledgements

We thank Thomas Kaufmann Jensen for demonstrating the Rational Rose tool on the Requirements diagram, and are very grateful to Arne Skou for filling in details about the model checking case study. We are also grateful to Heike Wehrheim for constructive comments on the draft of this paper.

8. REFERENCES

[1] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison Wesley, 1999.

[2] A.-P. Bröhl and W. Dröschel, editors. *Das V-Modell*, Software – Anwendungsentwicklung – Informationssysteme. Oldenbourg, 1995.

[3] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge, 1998.

[4] H. Dierks. PLC-Automata: a new class of implementable real-time automata. In M. Bertran and T. Rus, editors, *Transformation-Based Reactive Systems Development*, volume 1231 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 1997.

[5] H. Dierks. *Specification and Verification of Polling Real-Time Systems*. PhD thesis, Department of Computer Science, Univ. Oldenburg, January 2000.

[6] H. Dierks and C. Dietz. Graphical specification and reasoning: case study "generalized railroad crossing". In J. Fitzgerald, C. Jones, and P. Lucas, editors, *Formal Methods: Their Industrial Application and Strengthened Foundations (FME'97)*, volume 1313 of *Lecture Notes in Computer Science*, pages 20–39. Springer, 1997.

[7] C. Dietz. Graphical formalization of real-time requirements. In B. Jonsson and J. Parrow, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1135 of *Lecture Notes in Computer Science*, pages 366–385. Springer, 1996.

[8] S. Dupuy. RoZ version 0.3, 1999. Presented at *Formal Methods '99*. See also `http://www-lsr.imag.fr/Les.Personnes/Sophie.Dupuy/`.

[9] S. Dupuy, Y. Ledru, and M. Chabre-Peccoud. Translating the OMT dynamic model into Object-Z. In J. Bowen, A. Fett, and M. Hinchey, editors, *ZUM'98: The Z Formal Specification Notation*, volume 1493 of *Lecture Notes in Computer Science*, pages 347–366. Springer, 1998.

[10] R. France and B. Rumpe, editors. *UML'99: The Modified Modeling Language – Beyond the Standard*, volume 1723 of *Lecture Notes in Computer Science*. Springer, 1999.

[11] V. Friesen, A. Nordwig, and M. Weber. Object-oriented specification of hybrid systems using UML^h and ZimOO. In J. Bowen, A. Fett, and M. Hinchey, editors, *ZUM'98: The Z Formal Specification Notation*, volume 1493 of *Lecture Notes in Computer Science*, pages 328–346. Springer, 1998.

[12] R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors. *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*. Springer, 1993.

[13] K. Havelund, K. G. Larsen, and A. Skou. Formal modeling and analysis of an audio/video protocol: An industrial case study using UPPAAL. In *Proc. of the 18th IEEE Real-Time Systems Symposium*, pages 2–13. IEEE Computer Science Press, 1997.

[14] J. He, C. A. R. Hoare, M. Fränzle, M. Müller-Olm, E.-R. Olderog, M. Schenke, M. R. Hansen, A. P. Ravn, and H. Rischel. Provably correct systems. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 288–335. Springer, 1994.

[15] C. Heitmeyer and N. Lynch. The generalized railroad crossing: A case study in formal verification of systems. In *Proc. IEEE Real-Time Systems Symposium*. IEEE Computer Science Press, 1994.

[16] C. A. R. Hoare. Proofs of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[17] C. A. R. Hoare. Programs are predicates. In C. A. R. Hoare and J. C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, International Series in Computer Science, pages 141–154. Prentice-Hall, 1985.

[18] *IEC International Standard 1131-3, Programmable Controllers, Part 3, Programming Languages*, 1993.

[19] M. Joseph, editor. *Real-time Systems – Specification, Verification and Analysis*. Prentice Hall, 1996.

[20] S.-K. Kim and D. Carrington. Formalizing the UML class diagram using Object-Z. In R. France and B. Rumpe, editors, *UML'99: The Unified Modelling Language – Beyond the Standard*, volume 1723 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 1999.

[21] R. W. Lewis. *Programming industrial control systems using IEC 1131-3*. The Institution of Electrical Engineers, 1995.

[22] E.-R. Olderog, A. P. Ravn, and J. U. Skakkebæk. Refining system requirements to program specifications. In C. Heitmeyer and D. Mandrioli, editors, *Formal Methods for Real-Time Computing*, volume 5 of *Trends in Software*, pages 107–134. Wiley, 1996.

[23] P. Pepper and M. Wirsing. A method for development of correct software. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Constrcution of Correct Software*, volume 1009 of *Lecture Notes in Computer Science*, pages 27–57. Springer, 1995.

[24] A. P. Ravn and H. Rischel, editors. *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1486 of *Lecture Notes in Computer Science*. Springer, 1998.

[25] M. Schenke. Transformational design of real-time systems – part II: from program specifications to programs. *Acta Informatica*, 36:67–96, 1999.

[26] M. Schenke and E.-R. Olderog. Transformational design of real-time systems – part I: from requirements to program specifications. *Acta Informatica*, 36:1–65, 1999.

[27] B. Selic. Turning clockwise: Using UML in real-time domain. *Comm. ACM*, 42(10):46–54, October 1999.

[28] B. Selic and J. Rumbaugh. Using UML for modeling complex real-time systems. Technical report, ObjecTime, 1998.

[29] J. M. Spivey. *The Z Notation – A Reference Manual. Second Edition*. Prentice Hall, 1992.

[30] F. W. Vaandrager and J. H. van Schuppen, editors. *Hybrid Systems: Computation and Control*, volume 1569 of *Lecture Notes in Computer Science*. Springer, 1999.

[31] R. Wieringa. A survey of structured and object-oriented software specification methods and techniques. *ACM Computing Surveys*, 30:459–527, 1998.

[32] J. Woodcock and J. Davies. *Using Z Notation – Specification, Refinement, and Proof*. Prentice Hall, 1996.

[33] Z notation. Final Committee Draft, CD 13568.2, ISO Panel JTC1/SC22/WG19 (Rapporteur Group for Z), August 1999.
See `www.comlab.ox.ac.uk/oucl/groups/zstandards/` for the current status of the Z standardisation.

[34] C. Zhou, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.