Predictability in Real-time System Development

(2) A Case Study *

Jinfeng Huang, Jeroen Voeten and Piet van der Putten
Faculty of Electrical Engineering Eindhoven University of Technology
5600MB Eindhoven, The Netherlands
E-mail: J.Huang@tue.nl

Andre Ventevogel
TNO Industrial Technology
5600HE Eindhoven, the Netherlands

Abstract

In a companion paper [HVV04], we have investigated the deficiency of existing design approaches in supporting predictability for real-time control system development. A design approach is then introduced with an adequate support for efficient and consistent design and code generation. In this paper, this design approach is illustrated by a case study of a rail-road crossing system. The development process shows that our approach is able to preserve properties not only between design descriptions at different abstraction levels, but also between the design model and the generated implementation.

1 Introduction

In a companion paper [HVV04], we analyzed how to improve predictability in the design of complex real-time systems based on the semantics of development languages. Here we give a short summarization of the role of semantics during the system development.

1. The semantics of requirement languages: The semantics of requirement languages can contribute to verification and validation during the development stages 1. During verification, the satisfaction relation between design descriptions and requirements descriptions (properties) can be proven based on their semantics. During validation, especially the validation of an implementation against the design outcome, it is possible to predict the correctness of the implementation from properties of the model based on the formal linkage between the semantics of three development languages (requirement, design and implementation languages).

2. The semantics of design languages: The semantics of design languages should have sufficient support for compositionality (and composability). Both of them emphasize that components of a system can be treated as independent semantic units. As a result, abstraction/refinement of the system can be carried out in an efficient way, where each of its components can be abstracted/refined independently. Efficient abstraction/refinement can facilitate both top-down and bottom-up design paradigms.

*This research is supported by PROGRESS, the embedded systems research program of the Dutch organisation for Scientific Research NWO, the Dutch Ministry of Economic Affairs, the Technology Foundation STW and the Netherlands Organisation for Applied Scientific Research TNO.

1Although it is often unrealistic to formalize all requirements of the desired system in common practice, we believe, at least, that critical timing and safety requirements should be precisely specified.
3. *The semantics of development languages:* The semantics of implementation languages is often influenced by underlying platforms and are not always consistent with the semantics of design languages. To avoid misinterpretation of the design outcome, a (formal) linkage has to be established between the semantic domains of both design languages and implementation languages.

Based on the analysis of the role of the semantics during system development, we showed the semantic deficiency of existing development languages for real-time control systems and proposed a predictable approach to cope with this problem. In this paper, we will apply the proposed approach to the development of a rail-road crossing system. The development process shows that our approach is able to preserve properties not only between different design levels of abstraction, but also between the design model and the generated implementation.

The remainder of the paper is organized in six sections. In Section 2, the case study of the rail-road system is briefly introduced. Section 3 gives a comparison of two possible design solutions: a distributed and a centralized solution. Then the requirements of the system are analyzed in Section 4 based on the distributed control solution. Section 5 presents a serials of successive property-preserving design refinements. The automatic and property-preserving code generation is explained in Section 6. Conclusions and future work are given in Section 7.

2 A railroad crossing system

We choose a railroad crossing system as an example, which is inspired by the standard railroad crossing problem used to compare different formal methods for real-time systems [HJL93].

As shown in Figure 1, four stations are connected by two orthogonal tracks. Two trains (a and b) run back and forth between station 1 and 2 and station 3 and 4 respectively. Four sensors (A, B, C and D) are installed at some distance to the crossing to detect the passing of the trains. The correct operation of the system requires that 1) the system should never go into deadlock and 2) the trains should never collide and each train should spend as little time as possible on one journey from one station to the other.

Some relevant data are given as follows. In our example, the distance from each sensor to the nearest border of the crossing is 0.556 meters. The length of the crossing is 0.044 meters.

![Figure 1: The railroad crossing system](image-url)
The length of the train is 0.011 meters\(^2\). The upper bound of the transmission delay between the control of the system and the environment is 0.019 seconds. We assume that the speeds of the trains are constant between two sensor points (e.g. from point \(A\) to \(B\)) if no stop occurs. The speeds are 0.470 m/s for \(\text{TrainA}\) and 0.250 m/s for \(\text{TrainB}\). \(\text{TrainA}\) and \(\text{TrainB}\) need 0.045 meters and 0.015 meters respectively to decelerate before they stop.

### 3 A distributed control solution

In this example, we adopt a distributed control solution to design the system. Each (train or crossing) player is controlled by a separate and relatively simple process. The behavior of the whole system can be studied by investigating the interactions between these processes.

To obtain the distributed control, a critical zone is first defined as shown in Figure 1. When a train approaches a border of the critical zone from outside of this zone, its control process has to request for permission to enter the crossing. If the request is denied by the control process of the crossing, the control process of the train has to stop the physical train immediately and wait until permission is granted. To compensate for the deceleration time of the train and avoid the stopping of the train inside the crossing area, the critical zone is usually larger than the crossing area. When the train leaves the crossing, its control process has to release the crossing. As a consequence, the crossing is free and available for the other train. The benefits of such a design decision is that the complexity of each local control process remains almost the same if the physical system topology changes (e.g. the system is extended with more crossings and tracks). This is because the control process of a crossing is not affected by the topology of the system, and the control process of a train can adapt easily to the new system according to the number of crossings on its running track.

Another intuitive design solution is to use a centralized scheduling process to globally control the behavior of both trains. For example, based on the speed and the time of each train when it passes by the first sensor on each journey, the centralized schedule algorithm can calculate the time when each train arrives at the nearest crossing. Based on this information, the algorithm can determine whether a collision will happen. If it will, the process must let one of the trains stop for a certain amount of time to avoid the collision. However, in contrast to the distributed scheduling solution, the complexity of the centralized scheduling solution heavily relies on the global topology of the system. Especially when the topology of the system becomes complex, the scheduling algorithm itself can be extremely difficult to devise.

### 4 Requirement Analysis

The requirements of the system have been mentioned in the previous subsection. Now, we are going to specify these requirements more clearly so that they can be easily traced during the design process.

First, to guarantee that the trains should never collide, the following properties should be observed:

- The control process of a train should ensure that the train can enter the crossing area only if it obtains permission (property \(P_1\)).
- The control process of the crossing should ensure that permission to enter the crossing is assigned to at most one train at a time. (property \(P_2\))

\(^2\)In our example, the installed sensors can only detect the train when the middle part of the train passes by, so the length of each train plays a role in determining the time when the train arrives at the crossing. Furthermore, the length of the train also affects the time when the occupied crossing can be released.
Second, to maximize the efficiency of the system, the following parameters need special consideration.

- The size of the critical zone: A critical zone that is either too large or too small will lead to problems. A large critical zone can cause unnecessary waiting of the train. On the other hand, a small critical zone cannot provide sufficient space for the deceleration of the train. An optimal size of the critical zone can first be estimated and later adjusted.

- The releasing time of the crossing: Similarly, to ensure operation efficiency, the releasing time should not be too late or too early. Release the crossing too late might lead to the unnecessary waiting of another train, while a too early release may cause collisions. In principle, the control process of the train should release the crossing when the train just leaves the crossing border.

We can now specify in a quantitative way the timing constraints on the system behavior. For example, suppose the speed of the train ($s_{\text{train}}$) is 0.470m/s, its deceleration distance ($D_d$) is 0.045 meters, and the length of the train ($L_{\text{train}}$) and the upper bound of the sensor signal transmission delay ($t_{\text{sensor}}$) are the same as mentioned before. Then we can calculate the distance $D$ between the border of the critical zone and the border of the crossing area using the following formula:

$$D = s_{\text{train}} * t_{\text{sensor}} + L_{\text{train}}/2 + D_d$$

If the request for entering the crossing is denied, then the control process of the train should stop the physical train between $t_1$ and $t_2$ seconds after it received the first sensor signal on each journey (property $P_3$). In our case, $t_1$ and $t_2$ are 0.900 and 0.920 seconds respectively, based on the speed of the train and the distance between the sensor point and the border of the critical zone (such as point $A$ and $E$ in Figure 1).³

Therefore, we can see that the correctness of the system depends not only on avoiding granting access to the same crossing to both trains at a time, but also on the right time at which the control process issues the control commands, such as the time to request the crossing, the time to stop the train if the permission is denied and the time to release the crossing.

## 5 System design

In this section, we will use the top-down design paradigm to explore the design space of the railroad crossing system. The design process starts from building an abstract model of the system. Then successive refinements (an untimed refinement followed by a timed refinement) are carried out to obtain an adequate design model for the next code generation phase. In this way, the desired properties of the system can be effectively verified at different abstraction levels of the system. For example, property $P_2$ can be verified in the abstract model and property $P_3$ can be verified in the timed refinement of the model. Furthermore, properties verified at a high level of abstraction can be preserved in later refinements. For example, property $P_2$ verified in the abstract model is preserved in the refinement models.

### 5.1 The abstract model

In the abstract model, we first tackle the problem of obtaining a correct design for the control process of the crossing. The related requirements state that access to the same crossing should be granted to only one train at a time and that the whole system should be deadlock free. The state diagrams of involved players ($\text{Crossing}$, $\text{TrainA}$ and $\text{TrainB}$ ⁴) are shown in Figure 2, where

³If the train is stopped earlier than 0.900 seconds, the system may not be efficient enough. On the other hand, if the train is stopped later than 0.920 seconds, it might enter the crossing area incorrectly.

⁴$\text{TrainA}$ and $\text{TrainB}$ exhibit the same untimed behavior.
Figure 2a gives the state diagram of a possible design of the crossing control. To efficiently explore different design alternatives and avoid the disturbance from irrelevant details, the behavior of one train player (including the train control process and the physical train) can be modelled at a high level of abstraction. As shown in Figure 2b, the behavior of the train players is abstracted into a few states (4 states).

In Figure 3, a tool environment for making the model is demonstrated. The window (SHESim System Level Editor) in the middle of Figure 3 shows the main working space for the model construction. Here the abstract model consists of three processes, which are used to model a crossing player (Crossing) and two train players (TrainA and TrainB). In this tool environment, the behavior of each player can be succinctly modelled in the POOSL language. For example, the left part of Figure 3 gives a piece of POOSL code, which models the behavior of the train players. We can see that each state transition in Figure 2b can be directly mapped to a single POOSL statement. Furthermore, interactions between processes are accomplished through ports (such as crossing port) connected by static channels, such as channel A and B in Figure 3. The interaction diagram window at the right part of Figure 3 shows the interactions between the processes during simulation, which can assist designers to analyze possible design errors. The correctness of such a design solution can be verified by simulation of the abstract model.

Since the semantics of the modelling language is based on CCS [Mil89] with a timing and probability extension, it is also possible to verify properties of the model by using exhaustive verification techniques, when the system
5.2 The untimed refinement

In this model, the behavior of the train players (*TrainA* and *TrainB*) is refined by using two parallel processes: *Train_Image* and *Train_Actor* (as shown in Figure 4). *Train_Image* models the behavior of the control process of the physical train, while *Train_Actor* models the behavior of the physical train itself. The interactions between *Train_Actor* and *Train_Image* are also specified in the model. The code shown in Figure 4 describes a part of the behavior of process *Train_Image*, which starts from the moment it receives a message from the first sensor until it releases the crossing. The code in bold-face are refined actions w.r.t the abstract behavior of the train players in the last model. These actions are used to control *Train_Actor*. The refinement carried out in this step is performed in such a way that the observable communication behavior of the train players remains unchanged w.r.t. that in the abstract model. Therefore the semantics of the train players (*Train_Actor* || *Train_Image*) is a proper refinement of the train players in the abstract model. Properties satisfied in the abstract model also hold in this refined model.

\[
\begin{align*}
\text{train \ ? \ sensor;} & \quad \text{//sensor signaled } \\
\text{delay (T);} & \quad \text{// T: the time elapsed from the train passing the 1st sensor to reaching the border of the critical zone}\ \\
\text{crossing \ ! \ request;} & \quad \text{/}\ \\
\text{sel} & \quad \text{or} \\
\text{crossing \ ? \ granted;} & \quad \text{delay (T);} \quad \text{// T: the time elapsed from the train passing the border of the} \\
& \quad \text{critical zone to leaving the crossing area without stopping}\ \\
\text{or} & \\
\text{crossing \ ? \ denied;} & \\
\text{train \ ! \ pause;} & \\
\text{crossing \ ? \ granted;} & \\
\text{train \ ! \ resume;} & \\
\text{delay (T);} & \quad \text{// T: the time elapsed from the train restarting in the critical} \\
& \quad \text{zone to leaving the crossing area}\ \\
\text{les} & \\
\text{crossing \ ! \ release;} & \\
\end{align*}
\]

Figure 5: The timed refinement

model is at a high abstraction level.

Note that the timing information of the system has not been considered in this model.
5.3 The timed refinement

In this step, the desired timing properties should be investigated. For example, if the access request of the crossing from the Train_Imagel of TrainA is denied, we require that the Train_Actor passes the first sensor on each journey\footnote{This quantitative timing property is stricter than property $P_3$. The resulting model that satisfies stricter property can give more freedom for later code generation (see section 6).}. This can be formalized by using an MITL (Metric Interval Temporal Logic)\cite{AH93} formula $\square (r \rightarrow (p \rightarrow \Diamond [0.910,0.910]))$ where $\square$ and $\Diamond$ denote “always” and "eventually” respectively and $p, q$ and $r$ represent the following atomic propositions: $p :$ Train_Actor passes the first sensor, $q :$ Train_Imagel sends a $pause$ message to the Train_Actor and $r :$ Train_Imagel receives a $denied$ message from Crossing.

To be able to find a design solution satisfying the desired quantitative timing properties, we need first to incorporate the timing information into the untimed model. For example, the behavior description in Figure 4 can be extended with timing information as illustrated in Figure 5, where $T_1, T_2$ and $T_3$ can be calculated according to the relevant information, such as $T_1 = \frac{D}{S} - t_c - \Delta t$\footnote{$D$ represents the distance between the sensor and the border of the critical zone, such as from A to E in Figure 1. $S$ is the speed of the train. $t_c$ represents the transmission delay of the sensor signal. $\Delta t$ is used to compensate for the timing error caused by the small variation of the train speed. The value of $T_1$ is 0.910.}. After this extension, we can verify the desired quantitative timing properties on the timed model by exhaustive (model-checking) or non-exhaustive (simulation) techniques\cite{Gei02}. Since the model is relatively simple, we can check manually that the control part does satisfy the desired property.

5.4 The synthesis model

In the previous steps of the design process, design decisions are made through successive refinements and a detailed design model can be obtained during this process. Since the environmental part (Train_Actor) is not of our concern during the automatic code generation, we replace the Train_Actor components with an interface component (such as LEGO_Dacta in Figure 6) that establish the interface to the outside world.

In this synthesis model, Data class Dacta is embedded into the interface component, whose methods act as virtual interfaces to the outside world. For example, in Figure 6, the synthesis
model of the railroad crossing system is given, whose part behavior is demonstrated by the POOSL code. The code in bold-face is the data methods of an instance of the Dacta class, which facilitate the automatic code generation. It will be illustrated in the next section.

6 Code generation

In this phase, the synthesis model devised during the design process is automatically translated into an implementation. In order to generate a correct implementation, two important tasks should be carried out:

- **Implementation of the Dacta class:** The synthesis model provides the virtual communication interface (the methods of Dacta class) to the real world. To understand better how the virtual interface is implemented, we give a brief introduction here of the infrastructure of the system. In our example, a separate Dacta Controller provides the low level device drivers to control the LEGO motors and sensors. The designed control processes interact with the physical trains and sensors through the Dacta Controller (as shown in Figure 7). The control processes can communicate with the Dacta Controller through a serial port of a personal computer.

  Interface data class *Dacta* defined in the synthesis model only provides a virtual interface for control processes. The actual implementation of interface data class *Dacta* for the communication with the Dacta controller is provided by the Rotalumis tool. For example, the interface for stopping a train (*Dacta TurnOff(TrainA)*) can be implemented by using a segment of C++ codes as shown in Figure 8. The timing behavior of the interface im-

![Figure 7: The system infrastructure](image-url)

```c
static PDO *PDM_TurnOff(PDO **LV)
{
    unsigned char port BYTE buffer[2], DWORD numbytes;
    if (LV[1]~Class~PDC_Char)
        DisplayErr("The 1st Operand of dacta.TurnOff should be a Char.");
    if (dacta_running)
    {
        port~unsigned char LV[1]~4;
        if (port & port~9)
            buffer[0]=0x21; /* Turn off a motor*/
            buffer[1]=~1<<~(port~1);
            WriteFile(Comm. buffer. 2. &numbytes. NULL); /*serial port communication*/
        else
            DisplayErr("The bad port name in dacta.TurnOff");
    }
    return LV[0];
}
```

![Figure 8: The implementation of a interface data method](image-url)
Implementation should be consistent with the assumption of the transmission delay (Section 5.1) 9.

- **Estimation of the ϵ value:** The PET scheduler in Rotalumis ensures that the automatically generated implementation exhibits the same observation sequences as in the model. That is, the implementation satisfies the same qualitative timing properties (such as safety and liveness) as the model. At the same time, the PET scheduler also tries to synchronize the virtual time of each action in the model with the actual execution time in the implementation. The accuracy of the synchronization is often determined by the computational resource of the underlying platform 10. Since the accuracy of the synchronization has direct influence on the quantitative timing properties of the implementation, we need to estimate the upper bound of the timing deviations (the ϵ value) to predict the quantitative timing properties of the implementation. In practice, we can obtain the ϵ value in various ways: 1) One can run the implementation and let the scheduler record the timing deviation of each action at run time. Then the ϵ value can be estimated according to the recorded the timing deviations. 2) One can model the platform and the time duration of each action, then estimate the ϵ value in the integrated model of the platform and the system. More details about the estimation of the ϵ value can be found in [FVHC04].

![Figure 9: Measuring ϵ (the upper bound of timing deviations)](image)

In our example, we estimate the ϵ value 11 by means of the first way, due to the fact that it is difficult to model the underlying platform (Windows 98) adequately. We recorded around 300000 time deviations between the virtual time and the physical time during a 44-hour continuous running. As shown in Fig. 9b, most timing errors fall between 1 × 10⁻⁵ ~ 5 × 10⁻⁵ seconds, and only a very few time errors are around 3 × 10⁻⁴ seconds 12. These errors are actually caused by the background activities of the operating system. Therefore, the value of ϵ can be safely estimated as 0.001 seconds in our example.

Having this ϵ value, we can predict the quantitative timing properties of the implementation [HVG03] [HVG04]. For example, we know that property □(r → (p → ♦[0.910,0.910]q)) is satisfied in the model, so we can predict that □(r → (p → ♦[0.908,0.912]q)) will hold in our implementation. Note that this property is stronger than the required property P₃. This ensures that TrainA is never stopped inside the crossing area and that at the same time the system is operating efficiently. For certain platforms, the value of ϵ can be too large to ensure that TrainA always stops outside the crossing when a stop is necessary. In such cases, designers can either refine their model to satisfy stricter timing properties or adopt a platform with better performance [HVVvB04].

9In our example, the upper bound of the transmission delay is actually estimated according the implementation of Dacta.
10The time deviation between the model and the implementation is mainly due to the computation cost of actions occurring at a certain virtual time moment (as shown in Fig. 9a).
11A brief specification of the underlying platform of our implementation is PIII 700MHz, Memory 128M and Windows 98.
12Note that the timing deviations are observed after the issuing of the actions, i.e. the actual timing deviations should be less than this value.
Figure 10: The implementation of the system

7 Conclusion and future work

In this paper, we illustrated how to apply the proposed approach in a companion paper [HVV04] to developing real-time control systems. The development process shows that our approach is able to preserve properties not only between design descriptions at different abstraction levels, but also between the design model and the generated implementation. Nevertheless, a lot more work still needs to be carried out.

1. The $\epsilon$-hypothesis used to bridge the semantic gap between the design outcome and the implementation might be ineffective when expensive data computation operations are involved. Due to the possible large value of $\epsilon$, quantitative real-time properties of the implementation could be “far” away from the properties of the design outcome. A more suitable formal framework is needed to predict the properties of the implementation for such data-intensive systems.

2. In the companion paper [HVV04], we explained that the Rotalumis tool can automatically generate an implementation for a single-processor platform complying with the $\epsilon$-hypothesis. However, when a multi-processor architecture is used to deploy the implementation, a central PET scheduler does not exist. Development of a distributed scheduling mechanism that complies with the $\epsilon$-hypothesis is subject of future research.

References


