An executable interface specification for industrial embedded system design.

DOI:
10.1109/QSIC.2008.28

Document status and date:
Published: 01/01/2008

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

An executable interface specification for industrial embedded system design

Jinfeng Huang
Philips & LiteOn Digital Solutions Netherlands, Advanced Research Center,
Building SFJ-1 Glaslaan 2, 5616 LW Eindhoven, The Netherlands

Jeroen Voeten
Eindhoven University of Technology, Department of Electrical Engineering;
Embedded Systems Institute
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Serge Wolfs and Mark Coopmans
Philips & LiteOn Digital Solutions Netherlands, Advanced Research Center,
Building SFJ-1 Glaslaan 2, 5616 LW Eindhoven, The Netherlands

Abstract

Nowadays, designers resort to abstraction techniques to conquer the complexity of industrial embedded systems during the design process. However, due to the large semantic gap between the abstractions and the implementation, the designers often fails to apply the abstraction techniques. In this paper, an EIS-based (executable interface specification) approach is proposed for the embedded system design. The proposed approach starts with using interface state diagrams to specify system architectures. A set of rules is introduced to transfer these diagrams into an executable model (EIS model) consistently. By making use of simulation/verification techniques, many architectural design errors can be detected in the EIS model at an early design stage. In the end, the EIS model can be systematically transferred into an interpreted implementation or a compiled implementation based on the constraints of the embedded platform. In this way, the inconsistencies between the high-level abstractions and the implementation can largely be reduced.

1 Introduction

Nowadays, the industry has to deal with more and more complex embedded systems which involve multiple disciplines. For instance, the design of an optical device involves optics, mechanics, electronics and software. Among these disciplines, the software design plays an important role, because all these disciplines are tightly connected through software. The complexity of the system is directly reflected in the software design.

1.1 Problem statement

During the design of a system, it is often divided into components to manage the design complexity. Each individual component is usually easier to implement. This ”divide and conquer” concept decreases the design complexity for each component on one hand, but on the other hand it increases the communication overhead because of:

- communication between project members (explaining the functionality of each component);
- communication between components (method calls, synchronisations).

In practise, these communication overheads can cause problems and elongate the design process. Examples are misunderstanding between project members and mismatched method calls between components, which can be found at a very late design stage.

Model-driven approaches have been advocated in the past decade to remedy this problem encountered by the industry. Many concepts (such as interaction sequences and state diagrams) with powerful primitives (such as communication and parallelism) in this framework have been proposed to help designers to express their thoughts in a less ambiguous and a more succinct way. For instance, UML [1] provides more than a dozen diagrams to abstract different aspects of the system.

However, these concepts are not always effectively used in industrial design due to the lack of a smooth design trajectory. In a typical model-driven design approach used in

1550-6002/08 $25.00 © 2008 IEEE
DOI 10.1109/QSIC.2008.28

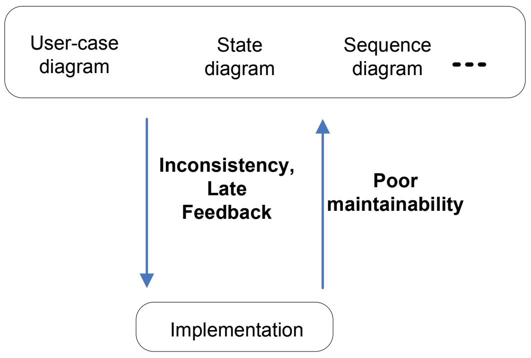


Figure 1: A typical model-driven design approach

the industry, the design process starts with defining static specifications (e.g. diagrams in UML) and continues with constructing implementations immediately (as illustrated in Figure 1). In this design process, designers often experience inconsistencies mentioned below.

- Each diagram only captures one static view of the system. Many diagrams (e.g. state diagrams, use-case diagrams, sequence diagrams) are needed to capture different aspects of the system dynamics. Much effort is required to maintain the consistency between these diagrams. Furthermore, these static diagrams do not provide sufficient analysis facilities to detect design errors in the dynamics of the system (esp. mismatched method calls, deadlocks, incorrect component interfaces etc).

- The static specifications are not consistent with the implementation due to the large semantic gaps. Static specifications can use a set of different primitives (non-determinism, synchronous/asynchronous communication, parallelism) to describe the behaviour of the system, which have no direct correspondences in the implementation language. Developers have to interpret these primitives based on their own understanding during the implementation. This unavoidably introduces additional design errors.

Due to the lack of clear linkages between different static diagrams and between the static diagrams and the implementation, the application of advanced concepts in the model-driven approach to embedded system design in the industrial environment is hampered.

1.2 Proposed solution

To reduce the inconsistencies that occur in a typical design process, we propose a design approach based on an executable interface specification (EIS in short). The proposed approach has the following characteristics.

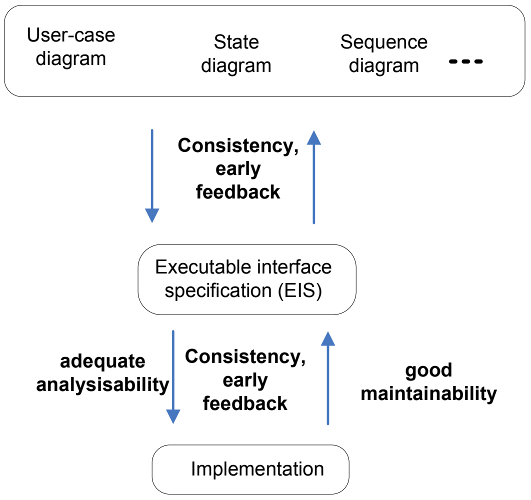


Figure 2: The proposed design approach

- The design process starts with interface state diagrams, which are similar to the commonly-used state diagrams but with specific semantics, which facilitates the transformation to an executable model. The interface state diagram is used to describe the architectural design at an early design stage.

- An executable model (EIS) is constructed from the state diagrams of components based on predefined rules. Consequently, the architectural design can be evaluated by verification or simulation techniques.

- The implementation can be derived from EIS systematically. Based on the constraints of the embedded platform, two different ways are illustrated to generate the implementation: compiled implementation and interpreted implementation.

The EIS model plays a core role in the approach (as illustrated in Figure 2). On one hand, it integrates different static specifications (e.g. interaction diagrams and state diagrams) into one unified dynamic specification. Many inconsistencies between different static specifications can be immediately observed during the execution. Designers can obtain the feedback of the interface design at an early design stage. On the other hand, it has to be transferred into the implementation consistently. To achieve this, a systematic transformation mechanism has to be available, which bridges the semantic gaps between the EIS model and the implementation.

The remaining part of the paper is organised as follows. Section 2 looks into each step of the proposed design method and illustrates the consistencies between these steps. Section 3 applies the proposed approach into the design of an embedded system: the control software of the tray

S::=	E	expression
	$m(E_1, ..., E_i)(v_1, ..., v_j)$	methods call
	par S_1 **and ... and** S_n **rap**	parallel composition
	S_1**;** S_2	sequential composition
	$ch!m(E_1, ..., E_i)\{E\}$	message send
	$ch?m(v_1, ..., v_i \| E_c)\{E\}$	(conditional) message receive
	sel S_1 **or ... or** S_n **les**	non-deterministic selection
	$[E_c]S$	guarded execution
	interrupt S_1 **with** S_2	interrupt
	abort S_1 **with** S_2	abort
	delay E	time synchronisation
	if E_c **then** S_1 **else** S_2 **fi**	choice
	while E_c **do** S **od**	loop
	skip	empty behaviour

Table 1: POOSL process statements

component in an optical device. Finally, section 4 gives a summary of the results.

2 EIS-based design approach

Different from existing executable specification languages / modelling tools such as SDL-2000[2], Rational Rose-RT[3] and TAU-G2[4], the proposed EIS approach aims at a general design trajectory for embedded system design in an industrial environment. We explicitly define the first abstraction of the system from which a design trajectory is presented. The idea of EIS can be integrated with / adapted to many of existing modelling languages. In the rest of the paper, POOSL (parallel object-oriented specification language)[5] is chosen as the specification language for the EIS model. In this section, a brief summary of the POOSL language is first presented. Then the proposed approach is introduced in details.

2.1 POOSL language

The POOSL language integrates a process part based on a timed and probabilistic extension of CCS [6] and a data part based on the concepts of traditional object-oriented languages [7]. A POOSL model consists of a set of parallel processes, which perform their activities asynchronously and communicate with each other synchronously by message passing. Each process can call and execute its methods which are formed by the statements in Table 1.

Here we give a brief explanation of the language to help in understanding the examples in the paper. More detailed information about the language can be found in [7] [8].

A POOSL model consists of a set of parallel processes connected by static channels. Each process has its own data space. It can only share its information with other processes through synchronous communication. Communications between processes are accomplished through ports connected by static channels. For instance, statement "out! request" indicates the willingness to send a request message through port "out" and "in? request" indicates the willingness to receive a request message from port "in". When the "in" and "out" ports are connected by a channel, both parts are synchronised and the communication is performed.

In addition to the parallelism between processes, a finer grain of parallelism (parallel activities) can be also specified inside a process using the **par** statement ("par S_1 and...and S_n rap"). Each activity can share a data space with other activities, and exchange its information with others through shared data.

The POOSL language provides the "delay" primitive to specify timing information in the model. Similar to many other formal languages, the timing semantics of POOSL relies on the two-phase execution model in [9]. The state of the model can change either by asynchronously executing actions (Phase 1) or by synchronously consuming time (Phase 2). Time advances only when no action can be performed. This timing semantics assumes that actions are instantaneous in the model, which largely simplifies the analysis of the timing behaviour.

In addition to the traditional "if" statement, non-deterministic selection ("sel S_1 or...or S_n les") does not specify conditions for its branches. This facilitates designers to abstract system behaviour. This is due to the fact that there are not always enough details available to determine the conditions when making an abstraction of a system behaviour or one component has no knowledge of its peers' behaviour.

For instance consider a system consisting of a consumer and a producer. The consumer first requests for the product from the producer. If the producer is not empty, the consumer takes one product and consumes it within 2 time units. Otherwise, the consumer waits for 0.1 time units and checks the availability of the producer. The above mentioned behaviour is endlessly repeated. Such a behaviour can be specified by three process methods in POOSL given in Table 2.

The Main()() method consists of a parallel structure where Consumer()() and Producer()() are specified as two parallel activities. The Consumer()() and Producer()() methods specify the behaviour of the Consumer and the Producer respectively at an abstract level. In the specification, ToPro (FromPro) and FromCon (ToCon) are a pair of connected ports. The Producer's internal behaviour is abstracted by the non-deterministic structure (sel), which facilitates a fast evaluation of design ideas. Note that the infinite behaviour of the Consumer and the Producer are specified by the tail recursion.

```
Main()()
  par
    Consumer()()
  and
    Producer()()
  rap.

Consumer()()
  BOOL fready;
  ToPro ! request;
  FromPro ? ack (fready);
  if (fready) then
    delay 2
  else
    delay 0.1
  fi;
  Consumer()().

Producer()()
  FromCon?request;
  sel
    ToCon!ack(TRUE)
  or
    ToCon!ack(FALSE)
  les;
  Producer()().
```

Table 2: The POOSL model of the consumer and producer example

In addition to the primitives mentioned above, the POOSL language offers a set of powerful primitives such as Guard, Interrupt and Abort primitives to facilitate designers to express their thoughts in a succinct way. In the later subsections, we only need to take use of a subset of these primitives to specify interfaces of the component, which also gives more freedom during the transfer from an EIS model to its implementation.

2.2 Interface state diagram

In an embedded system, components interact with each other through their interfaces. These interfaces play an important role during the system design which specify the behaviour of a component in an arbitrary environment. On the one hand, these interfaces offer a natural abstraction to understand the component behaviour. On the other hand, they are the skeleton for the later implementation.

However, due to the ambiguous semantics of the interfaces and their interactions in a typical design approach, a proper way is lacked to analyse the system behaviour based on them. For instance, the interfaces of a component can be

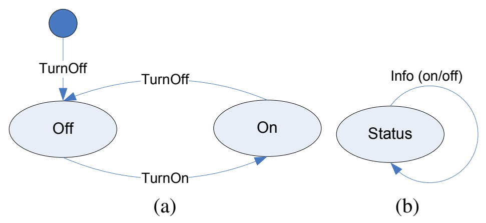


Figure 3: Parallel state diagrams of a LED component

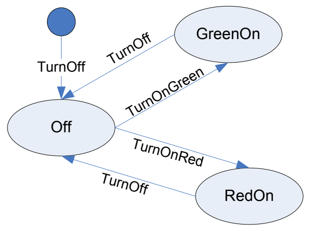


Figure 4: The state diagram of a two-colour LED component

interface functions and the interactions between the components can be function calls. The precedence relations between interface functions are not explicitly specified. For instance consider a control component which has four interface functions Start, Speedup, Slowdown and Stop. The Speedup function might only have to be called after the Start or Slowdown, but this kind of relations are normally missed in the interface specification. Consequently, it is difficult to detect improper function calls from the environment in a systematic way.

In our approach, we consider that each component has a set of states. Components interact with each other through communication, synchronisation or time synchronisation (interface interactions). The behaviour of each component can be specified by interface state diagrams, which has the following elements:

- Filled circle (●) represents the initial state of the component.
- Circle (◯) with a label denotes a state of the component.
- Arrow (→) denotes a state transition or a time delay. The name of communication/delay (if any) causing the transition is added to the arrow body.

It can be seen that the interface state diagram uses a subset of the notations of the state diagram defined in UML.

For instance, consider a LED component, which offers two states to its environment: On and Off. In the state diagram shown in Figure 3, each state transition represents a synchronisation with its environment.

2.3 The EIS model

During the design of an industrial embedded system, the system architecture is usually defined first where the system is divided into components and the interfaces of the components are defined. Due to the lack of a proper approach and tool, the quality of the architecture design heavily relies on the wisdom and experience of the architects. Especially when the system becomes more and more complex, many errors in the architecture design can only be found at a very late integration stage, which leads to time-consuming and costly design iterations.

The purpose of the EIS model is to verify the dynamic behaviour of the system based on component interfaces and to give early feedback on the correctness of the architecture design. In the following, several rules are proposed to construct an EIS model in POOSL language from interface state diagrams.

- ***Rule 1***: Each state in the diagram is specified as a process method in the EIS.
- ***Rule 2***: The state transitions with labels are mapped directly to synchronous communications between components or delay statements.
- ***Rule 3***: Each state in the diagram can have more than one outputs. These outputs are grouped by the sel structure in the EIS model.
- ***Rule 4***: Parallel interface state diagrams are transferred into parallel activities.

The notions behind ***Rule 1*** and ***Rule 2*** are rather straightforward. Here, some further explanation of ***Rule 3*** and ***Rule 4*** is given.

As mentioned earlier, each state in the diagram represent a state of the component. The output edges represent different possible execution traces of the component from that state. The choice of the future possible traces relies on the environment or implementation details of the component. It is not determined at the interface design stage yet. Therefore a non-deterministic structure (sel) is used to specify all potential consequences.

In the POOSL language, there are two forms of parallisms: parallel processes and parallel activities. These parallel interface diagrams are closer to the parallel activities concept, since parallel interface diagrams usually share data. Therefore, these parallel interface diagrams are transferred into a group of parallel activities in the EIS model.

Consider the Off and Init states of the LED component mentioned previously. In the EIS model, They can be specified as follows by using the proposed rules. The component switches to the On state when it receives a TurnOn message from its environment.

```
Off()()
   led ? TurnOn;
   On()().

Init()()
   led ? TurnOff;
   Off()().
```

Parallel state diagrams in Figure 3 are specified as follows.

```
par
    Init()()
and
    Status()()
rap .
```

Now we assume that the LED light offers a little bit more complex functionality. It can be lighted with different colours (e.g. red and green). Then its state diagram can be as shown in Figure 4. In this case, the corresponding process method of Off is constructed as the following based on the proposed rules.

```
Off()()
   sel
      led ? TurnOnGreen;
      GreenOn()()
    or
      led ? TurnOnRed;
      RedOn()()
   les.
```

It can be seen that the mapping from the interface state diagrams to the EIS model is rather straightforward. Each element in the state diagram is one-to-one mapped to a POOSL statement. Since the POOSL language has an operational semantics, it allows the EIS model to be simulated or verified [7].

2.4 Implementation framework

In the EIS model, the interactions between components and their environment are evaluated. However, there is still a gap to fill to obtain the final implementation. Depending on the constraints of the embedded platform, two ways to transfer an EIS model to its implementation are introduced.

2.4.1 Interpreted Implementation

The EIS model specifies the system behaviour at a rather high abstraction level. To be able to transfer design ideas expressed in the EIS model to an implementation for a specific platform, we still need refine the EIS model with more details. In [10], a systematic way has been introduced to refine a model consistently and a transformation mechanism to generate correct implementation by construction. The expressiveness of the modelling language has a direct impact on the transformation mechanism. As mentioned earlier, some notations in the modelling language may have no correspondences in the implementation language. It is not always feasible to make a direct map between syntactic structures in the modelling language and those in the implementation language. In the approach proposed in [8][10], this gap is bridged by introducing VMs (virtual machines) and a central scheduler. Notations from the modelling language are interpreted according to its semantics in VMs during the run-time. The behaviour consistency between the model and the implementation is formally proven in [11]. This approach can significantly shorten the design time and improve the implementation quality [12].

Embedded systems are typically resource-constrained however. For instance, a low cost chip (MT1926) is often used in the manufacturing of Blu-Ray drivers, which only provides a group of 64K memory modules on board. Each piece of code has to be manually assigned to a specific memory module. Since the VMs and the central scheduler are transparent to the designers in the approach, the designers have less flexibility to tune their code to fit in resource constrained platforms. In our case, the scheduler and the VMs in [8][10] requires around 200K memory. Therefore, the approach proposed in [8][10] needs to be further adjusted to be applied for these platforms.

2.4.2 Compiled implementation

To give designers more freedom to manipulate their code to fit in a specific platform, and at the same time to allow designers to evaluate the architectural design at an early stage, we can take use of a subset of primitives provided by the POOSL language to specify the EIS model. As explained in the subsections, an EIS model only contains synchronise communications, time, limited parallelism [1] and nondeterminism at the process level. The remaining process level primitives such as guard, interrupt and abort are not used in EIS. On the one hand, this decision sacrifices certain expressiveness power of the modelling language, but on the other hand, it provides the opportunities to simplify the scheduler and to eliminate the VMs in the implementation. In the following, a sketch of the transformation from the EIS model to its implementation is discussed.

In the implementation of the EIS model, there is a central scheduler, which schedules the execution of each component in the system. Each component uses a special variable to memorise the current state for each of its parallel activity. Furthermore, since at each state, the component may have several possible execution traces, a component keeps a waiting list which stores all waiting actions (sending communication, receiving communication or delay) at the current state and a ready action which records the selected action from the waiting list by the scheduler. The scheduler sets ready actions for components by first checking the match of sending and receiving communications from different components. If there is no matched communications, the delay expiration of each component is checked [2]. After a ready action of a component is set, the scheduler invokes a special function (Run) of the components, where the component behaviour is progressed based on the ready action and the current state. The component stops its execution until the moment that new actions are inserted into the waiting list or that it reaches its end state. In this way, the interaction behaviour in the EIS model can be preserved into the implementation.

[1]One component can have more than one parallel interface state diagrams and different components are in parallel.

3 Case study

In this section, the proposed method is illustrated by designing a tray component in an optical device (e.g. a DVD player). The component allows users to open and close the physical tray in the system through various ways (e.g. manual operations, wireless connections, or software applications). Furthermore, the component should have the capability to deal with some unusual scenarios, e.g. the tray is blocked on the half way of the movement because of obstacles in the environment. Figure 5-a illustrates briefly the interaction scenarios between the Tray component and its environment. The corresponding interface state diagram is given in Figure 5-b, where the component has six states. At the initial state, the component is ready to receive the Move_in message from its environment. Then the component goes into the Moving_in state, where it has three possible consequences. The tray may open completely where the component reaches the In state. The tray may receive a Move_out message during the moving, then the component goes to the Moving_out state. It is also possible that the tray is blocked in the middle, where the component goes into the Unknown state. Similarly, the remaining behaviour of the component is specified in the interface state diagram.

To evaluate the correctness of the architectural design, the state diagrams are converted into an EIS model (as shown in Figure 6. The conversion can be done according to the rules introduced in Section 5. For instance, When the component reaches the Moving_in state, its behaviour can be specified as:

[2]Note that, when a ready action is set for a component, the waiting list of the component should be emptied. This is due to that fact that each action in the waiting list represents a possible trace of the component at the current state. If one action is chosen by the scheduler, a trace of the component is chosen and the remaining should be abandoned.

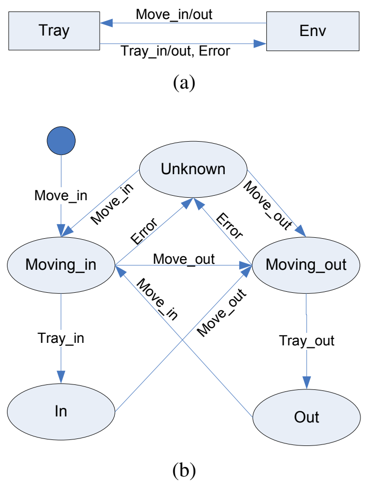


Figure 5: The state diagrams of the tray component

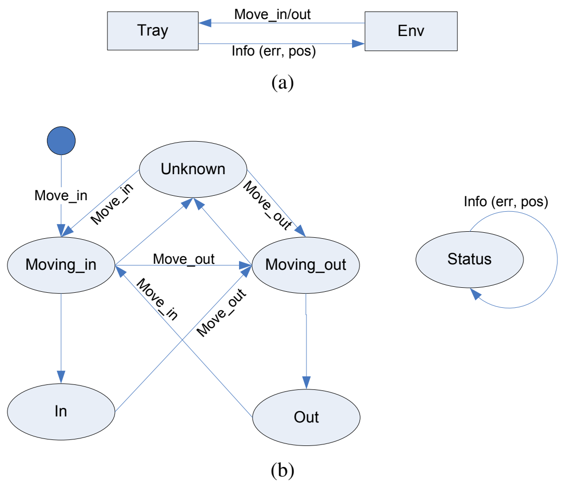
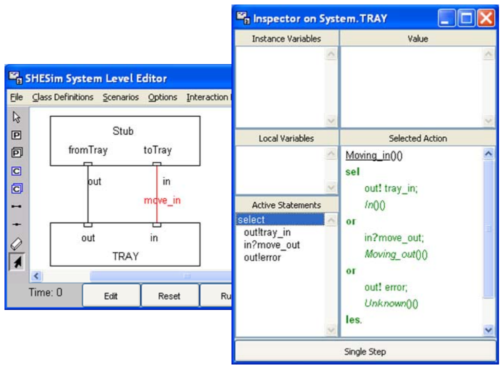


Figure 7: Another design of the tray component



```
select
  out!tray_in
  in?move_out
  out!error
```

```
Moving_in()()
sel
    out! tray_in;
    In()()
or
    in?move_out;
    Moving_out()()
or
    out! error;
    Unknown()()
les.
```

Figure 6: The EIS model of the tray component

```
Moving_in()()
    sel
        out!Error; Unknown()();
    or
        in?Move_out; Moving_out()();
    or
        out!Tray_in; In()();
    les .
```

By simulating the EIS model, it can be observed that the component behaves as expected. However, in the first architectural design of the component, the behaviour of the component is specified as shown in Figure 7, where it has two parallel interface state diagrams. Env first checks the status of the tray component, which is represented by a pair elements (err, pos). err has four options NOERROR, NOINITIALIZATION, BLOCKED_IN and BLOCKED_OUT. pos has three options MIDDLE, IN_END and OUT_END. Env sends Move_in or Move_out based on the status of the Tray component. When this design is evaluated in the EIS model, it turns out that Env may obtain a ”stale” status of the component. For instance, When Env gets the status of (NOERROR, MIDDLE) from the tray component, it starts to perform its further actions according to status (NOERROR, MIDDLE). However, due to the parallel nature of the embedded systems, the tray component may continue move and go into the status of (NOERROR, IN_END) already at the same time. Furthermore, for the status representation, it turns out that NOINITIALIZATION is not necessary, BLOCKED_IN and BLOCKED_OUT can be combined as BLOCKED and IN_END and OUT_END can be combined as END. By taking use of the EIS model, different design solutions can be evaluated efficiently, which facilitates designers to obtain an optimal design solution at an early design stage.

4 Conclusion

In this paper, a design approach is presented for industrial embedded systems. The approach starts with a so-called interface state diagram which is similar to the existing state diagram concept, but with a specific semantics for a smooth transformation to the EIS model. A set of rules is proposed to consistently transfer interface state diagrams into an EIS model, where the architectural design can be evaluated. Taking use of a Tray component from an optical device driver, we demonstrate that the EIS model helps to find architectural design errors in an early design stage. Furthermore, it also provides a fast and cheap way to evaluate different design solutions. In the end, we discuss two ways to transform an EIS model to its implementation systematically. In this way, the large gap between the early architectural design (static diagrams) and the implementation can be bridged by the EIS model.

References

[1] S. W. Ambler, *The Elements of UML 2.0 Style*. Cambridge University Press, 2005.

[2] T. standardization sector of ITU, “Z.100 annex f1: Formal description techniques (fdt)–specification and description language (SDL),” Nov 2000.

[3] “Rational rose realtime,” http://www.rational.com/tryit/rosert/index.jsp, Feb. 2003.

[4] “Tau generation 2,” http://www.taug2.com/, Feb 2003.

[5] P. van der Putten and J. Voeten, “Specification of reactive hardware/software systems,” Ph.D. dissertation, Eindhoven University of Technology, The Netherlands, 1997.

[6] R. Milner, *Communication and Concurrency*. Prentice Hall, 1989, iSBN 0-13-114984-9 (Hard) 0-13-115007-3 (Pbk).

[7] M. Geilen, J. Voeten, P. van der Putten, L. van Bokhoven, and M. Stevens, “Object-oriented modelling and specification using SHE,” *Journal of Computer Languages*, vol. 27, 2001.

[8] L. van Bokhoven, “Constructive tool design for formal languages from semantics to executing models,” Ph.D. dissertation, Eindhoven University of Technology, The Netherlands, 2002.

[9] X. Nicollin and J. Sifakis, “An overview and synthesis on timed process algebras,” in *Proceedings of the 3rd Workshop on Computer-Aided Verification, LNCS 575*, A. K. G. Larsen, Ed. Alborg, Denmark: Springer-Verlag, July 1991, pp. 376–398.

[10] J. Huang, J. Voeten, and H. Corporaal, “Predictable real-time software synthesis,” *International Journal on real-time systems*, vol. 36, pp. 159–198, 2007.

[11] J. Huang, J. Voeten, and M. Geilen, “Real-time property preservation in approximations of timed systems,” in *Proceedings of 1st ACM & IEEE International Conference on Formal Methods and Models for Codesign*. IEEE Computer Society Press, 2003, pp. 163–171.

[12] J. Huang, J. Voeten, M. Groothuis, J. Broenink, and H. Corporaal, “A model driven approach for mechatronic systems,” in *Proceedings of IEEE International Conference on Application of Concurrency to System Design (ACSD)*. IEEE Computer Society Press, 2007.