Published in:
Dominik Haneberg, Gerhard Schellhorn, and Wolfgang Reif, editors, *FM-TOOLS 2002. The 5th Workshop on Tools for System Design and Verification*, pages 65–69, Augsburg, June 2002.
Universität Augsburg, Institut für Informatik. Technical Report 2002-11.

Test Development in Virtual Environments

Stefan Bisanz   Aliki Tsiolakis

University of Bremen, Germany
{bisanz,tsio}@informatik.uni-bremen.de

Abstract

In this article, a graphical specification method is proposed that aims to facilitate the development of formal test specifications. In order to concentrate on the knowledge about the application to be tested, domain experts who are not familiar with formal methods rather than test experts can create test specifications to be used for automated testing. By interaction with a virtual reality model of the system under test's functional interfaces, these specifications are intuitively developed and generated. Statecharts are used as formal specification language to represent the test specification. The generation of test specifications as well as the properties of the virtual reality model are discussed based on an example using the test specification tool VETS.

1 Introduction

The creation of system specifications for embedded real-time systems is a complex task and it is even more sophisticated to develop a formal and complete specification. Nevertheless, the benefits of formal system specifications are manifold. In particular, these specifications can be used to derive automatically test specifcations which can be used to automatically execute and evaluate the tests by an appropriate test tool. For example, *RT Tester*[1] supports automatic test generation, execution and evaluation based on formal test specifications. See [Pel98] for the theoretical background on specification based testing.

Our experience shows that most system specifications are informal and rarely complete or consistent. Thus, separate formal test specifications have to be developed manually based on the informal system specifications. This formalisation step requires considerable skills in the field of formal methods and thus the knowledge of test experts. In order to enable domain experts to create test specifications themselves, we propose an approach to generate formal test specifications by interacting with a virtual reality representation of the system under test (SUT). This so-called *Virtual Periphery* represents the functional interface of the SUT embedded into its environment. Interactions with the Virtual Periphery generate specification fragments that can be composed into complete test specifications. The interaction semantics can easily be learned since the user either interacts with objects representing elements in the real world (e. g., switches) or uses graphical menus, gesture interactions or speech input. The latter are enhancements used to distinguish different interaction modes and concepts like stimulus or reaction specification, alternative reactions or parallelism.

[1]The RT Tester tool has been developed by Verified Systems International GmbH (http://www.verified.de) in cooperation with University of Bremen and has been applied in many application areas – from avionics to railway controllers. See for example [PT02].

For example, consider the set of no-smoking signs in an airplane that can either be switched on/off manually using a specific switch in the cockpit or automatically depending on the state of several sensors. To specify that the signs are switched on/off manually (and thus to generate the necessary mapping between the signs and the switch state), the user has to interact with the switch in the cockpit as well as with the signs. To simplify the example in the following, we will trim our setting to (a set of) no-smoking signs and a two-state switch and will focus on the manual switching of the signs' state.

Nevertheless, the Virtual Periphery cannot depict all concepts which might be necessary to develop complete test specifications. For example, the Virtual Periphery can only reflect the present point in time while the specification process might necessitate to investigate past interactions or potential future events. It is a straight-forward solution to use the representation of the formal specifications for the additional view on the test specifications. Hence, the formal specification language has not only to be expressive enough for testing purposes but also intuitively understandable. In this paper, we focus on statecharts which provide a graphical representation as well as a formal semantics. Note that we will not discuss specific statechart semantics. A comparison is provided by [vdB94].

This paper gives an overview how test specifications represented as sets of statecharts can be developed by interactions with the Virtual Periphery. This specification method is supported by the specification tool VETS (Virtual Environment for Test Specifications) that is currently developed by University of Bremen. To give more evidence to our descriptions, we will focus on the no-smoking example depicted above.

2 Interface Objects

Embedded into non-functional geometry that serves to model the SUT's environment, interface objects constitute the functional part of our Virtual Periphery. They consume input from and provide output to the SUT's operational environment. Each interface object has several possible states, and state changes can be initiated by user interactions within the Virtual Periphery. Figure 1 shows a two-state toggle switch with possible states ON and OFF that is currently *directly manipulated*: the movement of the 3D cursor's fingertips chooses which part of the toggle switch is pressed down and therefore triggers the corresponding state change. Associated with each state change is the input event SWITCH.SET_ON or SWITCH.SET_OFF, respectively. Other examples of interface objects are signs that produce output and touch screens that combine input and output.

Figure 1: 3D cursor interaction with toggle switch

In order to group similar interface objects, each interface object is associated with specific attributes and attribute values can be used to select a set of them. Consider for example the no-smoking signs in an airplane which have the attributes TYPE, SEAT ROW and AISLE. The constraint `type = no-smoking sign and seat row = 17 and aisle = right` selects all no-smoking signs in seat row 17 in the right aisle whereas the constraint `seat row = 17 and aisle = right` selects interface objects of different types, i. e., all no-smoking signs as well as all other

signs (e. g., fasten-seatbelt signs) and buttons (e. g., pax-call buttons) located at seat row 17 in the right aisle.

For each interface object type, the set of attributes as well as the possible states, state changes and the geometry is shared by its interface objects. Thus, interface object templates can be provided by means of libraries. Part of each interface object template is a *behaviour template* – a statechart representing the possible states and state changes. Figure 2 shows the behaviour template of the switch from figure 1. It contains the two states ON and OFF and two transitions, triggered by the events SWITCH.SET_OFF and SWITCH.SET_ON, respectively that reflect the possible state changes.

A similar behaviour template is associated with a sign, see figure 3, that keeps track of the sign's actual state. But as the sign provides outputs to the SUT's environment and therefore shows the SUT's reactions, the behaviour template furthermore contains a checker component that checks if the sign's state changes occur appropriately depending on certain events or conditions within the SUT. It distinguishes valid situations which are denoted by the states DARK_CHECKED and LIGHT_CHECKED, delaying situations that are expressed with the states DARK_NOT_YET, LIGHT_NOT_YET and pending situations that are represented by DARK_PENDING and LIGHT_PENDING. In pending situations, an event or condition to change the sign's state has occured but the sign has not reacted correctly so far. In delaying situations, the sign is not expected to react yet. Valid situations denote that the reaction of the sign has already been correct. Figure 4 shows the checker template of the sign as well as further transitions and labels to be added during the specification process. In particular, these are the events and conditions that trigger the state changes.

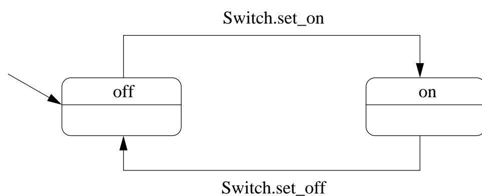

Figure 2: Switch statechart

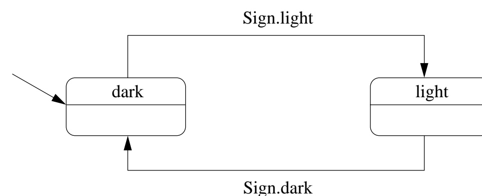

Figure 3: Sign statechart

3 Interactive Test Specification Development

Given the behaviour templates of the switch and the sign, the objective of a simple test specification could be to check the following condition:

```
The sign must be lighted, if the switch is on, else it must be dark.
The maximum state change delay is t time units.
```

It is assumed that the switch is initially in state OFF and the sign is initially DARK. In the sign's checker template, the application specific dependencies between switch and lamp are not yet defined. The following steps describe how the dependencies are introduced into corresponding behaviour templates for the concrete application context. With respect to the current specification context, the system state has to be consistent in order to specify only correct behaviour. Needed Virtual Periphery features are introduced as they are used in this example. See [BKLP01] for an overview about 3D interaction and [Shn83] for a general discussion of direct manipulation.

1. First, the Virtual Periphery's interaction mode is set to TIMED by applying the so-called *time gesture*. Thereby the 3D cursor which is equipped with a watch is

rotated as if the user is taking a look on it. The effect is that further interactions are time dependant such that expected reactions must occur within a specific time interval.

2. Now the specification context is chosen by using the so-called *reference gesture* and then touching the sign. Thus the sign is the context object for the following interactions and generated fragments will be added to this sign's behaviour specification that is predefined by its behaviour template. The reference gesture is a rotation of the 3D cursor so that its palm is directed upwards. Navigation to the sign's location can be accomplished by previously issuing the *voice command* VIEW SIGN.

3. To define the trigger for the state change of the context object, the switch is directly manipulated with the 3D cursor to its state ON. Again, a previous navigation to the switch by voice command may be needed.

4. The effect of the trigger (i. e. the state change of the context object) is selected using a *3D menu*. That is a graphical menu which is embedded into the Virtual Periphery and provides a state selection method for interface objects that are not targets of tactile interaction in the real world and that therefore cannot be directly manipulated in the virtual world. The chosen state LIGHT is considered as the target state of a transition, whereby the sign's current state DARK denotes the source state. But as the checker behaviour is specified, actually the transition from DARK_CHECKED to LIGHT_NOT_YET is affected. It is labelled with the guard IN(SWITCH.ON) that represents the condition where the switch has entered the state ON. Note that this is the trigger defined in step 3. Furthermore, the timed mode that was activated in step 1 takes effect. The lower bound of the reaction's time interval is reflected by an appropriate timeout event that is applied to the transition from LIGHT_NOT_YET to LIGHT_PENDING. A new transition from LIGHT_NOT_YET to LIGHT_CHECKED can fire before this timeout event occurs, denoting the premature occurence of the sign's reaction and thus of an error. The upper bound is checked by a new transition from LIGHT_PENDING back to LIGHT_PENDING. It is labelled with a timeout event that occurs as soon as the upper bound is reached. The firing of this transition denotes an error, because LIGHT_CHECKED was not reached so far. Default values are used for the lower and upper bounds. They have to be further adjusted (or omitted), e. g., by the use of a 3D menu. The correct values for our example condition are `t1=t3=0` and `t2=t4=t`.

The second part of our specification is created similarly to the previous steps. Of course, the affected states and labels have to be replaced correlatively. Figure 4 shows the resulting sign checker statechart. Note that the Virtual Periphery's selection facility enables the extension of our example condition to `iff the switch is on, then signs` S_1 `...` S_n `will be lighted`.

4 Conclusion

In this article, an intuitive way to develop formal test specifications has been introduced. Domain experts who are not familiar with formal specification languages interact with a virtual reality representation of the SUT and thus create simple statechart specifications without a detailed understandig of the underlying concepts. Of course, some basic

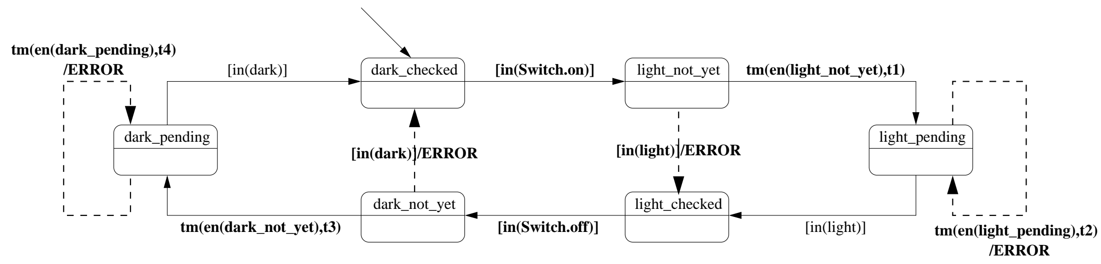

Figure 4: Extended sign checker statechart. Plain text and solid lines denote template components. Bold text and dashed lines represent interactively added components.

understandig of concepts like sequencing of events and parallelism is required, but our approach can support improvement and training while the domain expert creates more elaborate specifications and therefore gets familiar with formal specification languages. Within a test team our specification method can be applied by domain experts to create basic test specifications that are then composed and refined in cooperation with test experts.

The Virtual Periphery is implemented by the specification tool VETS, using Java, Java3D and VRML. It can be customised and connected to external tools in order to flexibly exchange the formal specification language. To develop statechart specifications, ArgoUML[2] is used. Since the developed test specifications are used with RT Tester that accepts test specifications as LTS (labelled transition systems), any formal specification language that can be translated to LTS can be used.

In future work, we will further improve the Virtual Periphery concept and proceed with the development of the corresponding VETS tool. As well, we will evaluate the consideration of further formal specification languages. For example, hybrid automata enable modelling of mixed discrete and continuous behaviour and will be investigated with respect thereof.

References

[BKLP01] D. Bowman, E. Kruijff, J. LaViola, and I. Poupyrev. An Introduction to 3D User Interface Design. *Presence: Teleoperators and Virtual Environments*, 10(1):96–108, 2001.

[Pel98] Jan Peleska. Testing reactive real-time systems. Tutorial, held at the FTRTFT '98, Denmark Technical University, Lyngby, 1998. Updated revision. Available as http://www.informatik.uni-bremen.de/agbs/jp/papers/ftrtft98.ps.

[PT02] J. Pelska and A. Tsiolakis. Automated Integration Testing for Avionics Systems. In *Proceedings of the 3rd ICSTEST – International Conference on Software Testing*, April 2002.

[Shn83] Ben Shneiderman. Direct manipulation: A step beyond programming languages. *IEEE Computer*, 16(8):57–69, 1983.

[vdB94] M. von der Beeck. A comparison of StateCharts variants. In *Proc. of Formal Techniques in Real Time and Fault Tolerant Systems*, pages 128–148, Berlin, 1994. Springer-Verlag.

[2]ArgoUML is a freely availabe open source tool, see http://www.argouml.org.