Published in:
Rob Hierons and Thierry Jéron, editors, *Formal Approaches to Testing of Software. FATES'02. A Satellite Workshop of CONCUR'02.*, pages 121–135. INRIA, August 2002.

Using a Virtual Reality Environment to Generate Test Specifications

Stefan Bisanz Aliki Tsiolakis

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

Abstract

The creation of test specifications that can be used for automated testing requires considerable skill in the field of formal methods. This article proposes a method that enables the development of test specifications by interaction with a virtual reality representation of the system under test. From these interactions, a formal test specification is generated. Its goal is to reduce the need for formal methods expertise and therefore concentrates on the knowledge of the application to be tested. It addresses domain experts who are not familiar with formal methods.

In this article, the character of the virtual reality model as well as its creation are discussed. Further, the generation of test specifications is explained. Statecharts are used as formal specification language for the result of the generation step.

1 Introduction

Testing embedded real-time systems is a complex and time-consuming task and thus a high level of automation is advisory. Ideally, the test team receives a complete and consistent system specification in a formal specification language. By use of a test tool, the automatic generation of test cases, the automatic execution of the test and the automatic evaluation can be supported based on this specification. This idea relies on two main preconditions: On the one hand, the existence of a formal system specification developed by the domain experts and, on the other hand, on the availability of an appropriate test tool.

However, most system specifications are informal, natural language descriptions of the system's behaviour and the test experts have to develop the necessary test specifications manually based on these system specifications. Since the formal specifications require considerable skills in the field of formal methods, the domain experts can usually not generate the test specifications themselves. Nevertheless, approriate

test tools can be found. For example, *RT Tester*[1] supports automatic test generation, execution and evaluation based on formal test specifications in Timed CSP. These Timed CSP specifications are then translated into labelled transition systems (LTS). RT Tester has been applied in many application areas – from avionics to railway controllers.

Thus, the process of automatic test execution and evaluation is sufficiently supported by existing test tools. To fill the gap, we want to introduce a new approach of generating formal test specifications by interacting with a virtual reality representation of the system under test (SUT) called the *Virtual Periphery*. The Virtual Periphery represents the functional interface of the SUT embedded into its environment. For example, consider as a SUT the fasten-seatbelt signs in an airplane that can be switched on or off automatically depending on the status of specific sensors or manually by a dedicated switch in the cockpit. The functional interface consists of interface objects like signs, switches or sensors which are represented as interactive objects in the Virtual Periphery. The additional non-interactive objects constitute the geometry that is not of functional relevance to the SUT but serve to model the SUT's environment. To specify that the fasten-seatbelt signs are switched on by toggling the switch, one has to interact with the corresponding interface objects. Thus, the approach assists the domain experts intuitively during the specification process because the interaction's semantics can easily be learned. Interactions with VR components are mapped to specification fragments of a formal specification language that can be composed into the complete test specification.

Nevertheless, this basic Virtual Periphery is not sufficient to specify specific concepts, e. g., parallelism, or alternative interactions. For example, it is not possible to express by these interactions that all fasten-seatbelt signs have to be switched on in parallel because it is not possible to perform several simultaneous actions in the Virtual Periphery. To reduce this drawback, it is necessary to enhance the Virtual Periphery with visual representations of commands, graphical menues, gesture interactions, or voice commands. Furthermore, the Virtual Periphery only reflects the "present point in time" while it is not possible to investigate the history of past interactions or the potential future events. In order to overcome this, we have chosen statecharts as a formal but as well graphical specification technique. Statecharts provide an alternative view on the test specification fragments and facilitate their understanding. This choice has essential impact on our specification approach because, on the one hand, it affects the semantics of the VR interactions and, on the other hand, it defines the maximum expressive power of the resulting specifications.

In the next section, we give a detailed overview about the Virtual Periphery and some possible enhancements. The representation of the test specification fragments as statecharts is described in section 3. Section 4 refers to the integration of the generated statechart specifications with RT Tester. In section 5, we discuss ways to automatically generate the Virtual Periphery.

---

[1] RT Tester has been developed by Verified Systems International GmbH (http://www.verified.de) in cooperation with University of Bremen.

2 Virtual Periphery

The Virtual Periphery is an incomplete simulation model of the SUT, i. e., it is a model of the real world that only contains those objects that are relevant for the SUT's interface. Additional non-functional geometry enriches the Virtual Periphery and serves to model the SUT's environment such that it resembles the real world. The interface objects consume inputs (e. g., switches, buttons, sensors), yield output (e. g., signs, loudspeakers) or process input as well as generate output (e. g., touch screens, buttons with back light). Each interface object can be in a number of different states, and only specific state changes are possible. For example, a specific two-state toggle's states are ON and OFF, and a specific indicator can yield the states RED, YELLOW and GREEN. Each interface object is associated with specific attributes which include the interface object's type. Returning to our previous example, the fasten-seatbelt signs in an airplane denote a specific type of sign which has the attributes SEAT ROW and AISLE. Thus, specific objects of the same type can be identified. Furthermore, similar interface objects can be grouped, e. g., applying the condition `type = "Fasten-Seatbelt Sign" and seat row > 20 and aisle = left` identifies all interface objects of type *Fasten-Seatbelt Sign* in the left aisle of an aircraft at seat row 21 or higher.

As interface objects of the same type share geometry, possible states and attributes, interface object templates can be provided by interface object libraries. General purpose libraries contain general switches, indicators, etc. while domain specific interface objects (e. g., the fasten-seatbelt signs) are defined in domain specific libraries. The interface object templates and thus the libraries are modelled manually and are utilised within the Virtual Periphery creation process (see section 5).

The use of our Virtual Periphery exceeds simple simulation. By interacting with interface objects the user generates test specification fragments. This means that specific state changes in one input interface object are reflected in state changes of other output (or input/output) objects. More precisely, the state changes in the input interfaces are initiated by interactions and are used as stimuli for the SUT. The expected reaction is represented by the correct output.

Interaction with interface objects takes place by navigating a *3D cursor* that looks like a human hand. By touching an interface object (i. e., more exactly by colliding the 3D cursor with it), it gets selected for further interactions. Moving or rotating the 3D cursor generates a basic test specification element.[2] What movement or rotation is appropriate depends on the type of the selected interface object, but it should conform to the *direct manipulation metaphor*. This direct manipulation changes the interface object's state and its visual representation. For example, a toggle switch changes its state according to the rotation direction: the movement of the 3D cursor's fingertips chooses which part of the toggle switch is pressed down. Figure 1 shows a 3D cursor and three-state toggle switches in the cockpit of an airplane. For an introduction to 3D interaction see [BKLP01] and for a general discussion on direct manipulation see [Shn83] and [HHN86].

[2] If a conventional mouse is used for interactions, there are two dimensional mouse interactions that correspond to these three dimensional ones.

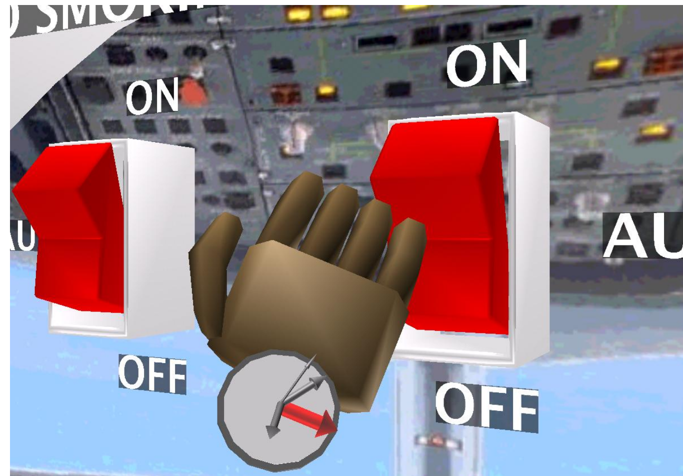


Figure 1: 3D cursor interaction with toggle switch

However, the direct manipulation metaphor is not appropriate for all interface objects. Consider interface objects that are not targets of tactile interaction in the real world (e. g., a sign or a sensor). Their state changes are selected using a graphical 3D menu within the Virtual Periphery that is triggered by a 3D metaphor of a mouse click, i. e., by a short movement of the 3D cursor's fingertips towards the interface object. An example of a 3D menu to select the state change of a sensor is given in Figure 2.

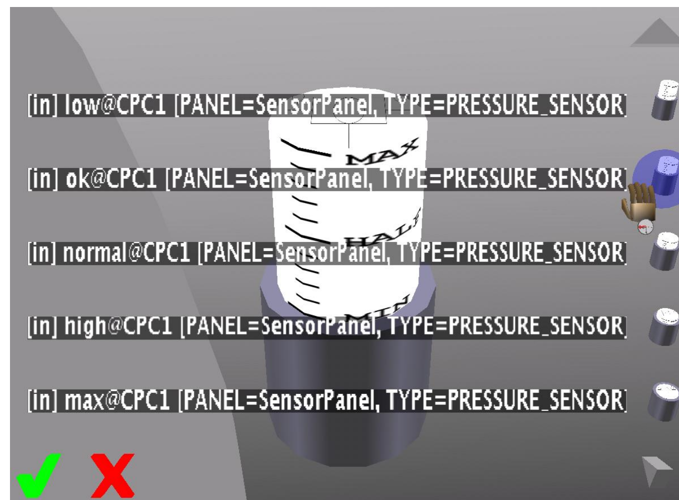


Figure 2: Selection of a sensor's state change using graphical 3D menu

Besides the central direct manipulation metaphor, we make use of a technique that integrates naturally within virtual reality: voice commands. While direct manipulation is used for interaction within the Virtual Periphery, speech input is used for *system control* and therefore for interaction with the Virtual Periphery itself.

In order to realize different semantics of direct manipulation interaction, the Virtual Periphery must provide different *interaction modes*. For example, a *causality* mode would enable specification fragments like

`if switch SW changes to state` ON`, then sign SI will be` LIGHTED,

and a *parallel* mode would enable

`all signs` SI_1 `...` SI_n `change to state` LIGHTED `in arbitrary order`

Note that speech based system control is superior to graphical system control because it does not interrupt the interaction within the Virtual Periphery. Simple *navigation* is also controlled by speech: predefined viewpoints can be activated such that the viewer[3] is immediately transported to the corresponding location within the virtual world.

Another feature supported by speech input is the *selection* of interface objects: in order to use several similar interface objects within a specification fragment, one selects them before interacting with one of them that then acts as a placeholder for all these objects.

While speech selection provides selection of currently visible interface objects, alternatives are mouse based selection on the one hand and attribute based selection on the other hand.

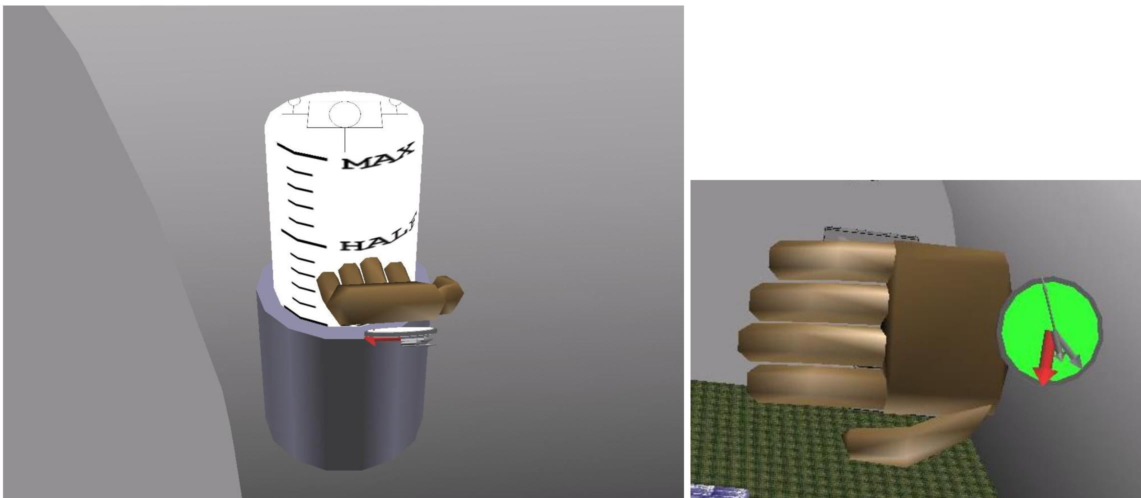


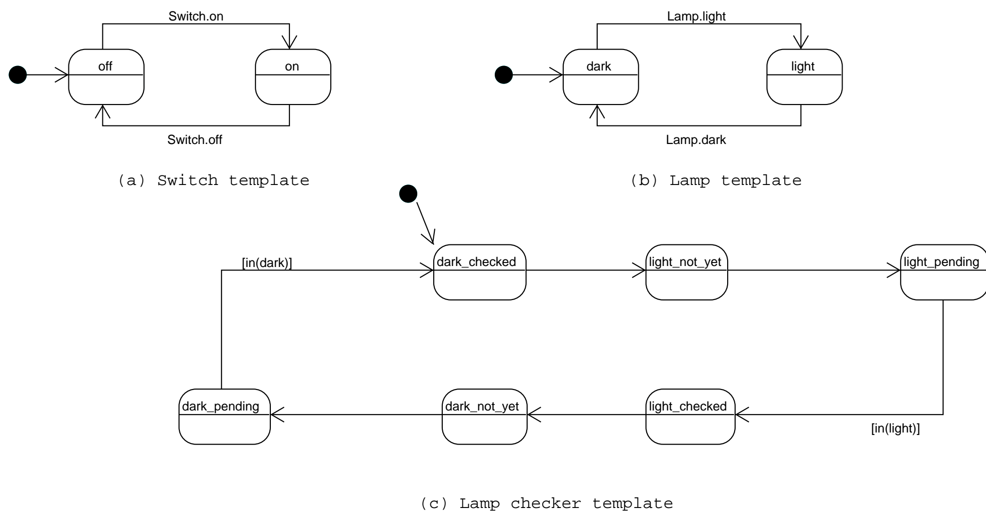
Figure 3: Reference gesture (left) and time gesture (right)

In order to switch specific interaction modes, *gestures* can be used. For example, the so-called *reference gesture* is a rotation of the 3D cursor so that it looks like an open hand, palm up. In combination with a subsequent direct manipulation interaction as described above, it references the complete interface object and defines the current specification context. The so-called *time gesture* will switch the specification mode so that further interactions are time dependant. Thereby, the 3D cursor which is equipped with a watch is rotated as if the user is taking a look on it. Both gestures are shown in figure 3.

[3]That is the person interacting with the Virtual Periphery.

3 Incremental Development of Statecharts by Interaction

Behaviour Template With each interface object a pre-defined *behaviour template* is associated: a statechart consisting of its possible states and appropriate transitions. The behaviour template of an interface object is created manually based on its interface and stored as an additional attribute of its template in the interface object library (see section 2). Note that the states in the behaviour template correspond to the informally described states of the interface object. Additionally, the behaviour template contains transitions between the states based on the interface description of the interface object. Considering a simple two-state switch as an example, the corresponding statechart contains two states ON and OFF. The state changes would be switching from ON to OFF and vice versa, therefore the statechart contains two transitions triggered by the events SWITCH.OFF and SWITCH.ON, respectively. Figure 4a shows the behaviour template of the two-state switch.



(a) Switch template

(b) Lamp template

(c) Lamp checker template

Figure 4: Behaviour template of switch and lamp

Additionally, the behaviour template can contain a checker component that is typically needed for our test purposes. As an example, consider a simple lamp or sign that provides outputs to the SUT's environment: while on the one hand the real lamp is in one of its states DARK or LIGHT (see figure 4b), we want to check if its state changes occur appropriately depending on certain events or conditions within the SUT. While these events and conditions are subject to the specification process, the checker component generally distinguishes valid and pending situations. Figure 4 shows the behaviour template of a lamp consisting of a statechart for the real lamp and the checker statechart. The latter is denoted in figure 4c and contains different states: The states DARK_CHECKED and LIGHT_CHECKED represent the correct (and checked) states and thus correspond conceptually to the states DARK and LIGHT in figure 4b. The states DARK_PENDING and LIGHT_PENDING represent the

situation when the event (or condition) to change the state has already occured but the correct reaction of the SUT has not been checked yet. Finally, DARK_NOT_YET and LIGHT_NOT_YET denote that after the occurence of the change event the SUT is not expected to react yet.

Interaction Scenario Let us now consider the interaction scenario for the following requirement:

```
The lamp must be dark, if the switch is off
and it must be lighted, if the switch is on.
```

The interface objects are a two-state switch and a lamp whereby the former is initially in state OFF and the latter is initially DARK. On template base, the application specific dependencies between switch, lamp and checker component are not yet defined. The following steps describe in a step-by-step manner how these dependencies are introduced to the templates of figure 4 for the concrete application context. Since only correct behaviour should be specified, the system state has to be consistent with respect to the current specification context.

1. The process is started by choosing the specification context, i. e., this is the lamp in the example. Therefore, we navigate to the lamp within the Virtual Periphery by issuing the voice command VIEW LAMP. By applying the reference gesture and touching the lamp, we reference the lamp's behaviour template (i. e., the corresponding statechart).
2. To define the trigger for the state change of the context object, we navigate to the trigger object, i. e., in this example to the switch, by voice command VIEW SWITCH. The following manipulation of the switch – a rotation of the 3D cursor – changes the state of the switch to ON. This state change is represented by a transition in the switch's statechart.
3. To define the effect of the trigger on the context object, the current state of the context object is considered as the source state for a transition. Since the lamp cannot be manipulated directly, the target state (i. e., the state LIGHT) has to be selected using a 3D menu. Although the corresponding 3D menu provides the states LIGHT and DARK (i. e., states defined in the lamp's statechart), the checker statechart is affected. The transition with source state DARK_CHECKED and target state LIGHT_NOT_YET is labelled with the guard IN(SWITCH.ON).
4. The same steps have to be applied for specifying the second part of our specification, respectively. Thereby, the transition from state LIGHT to target state DARK_NOT_YET is labelled with IN(SWITCH.OFF).

In the above specification scenario, the resulting specification fragment is only causally coherent but does not contain any timing constraints. Moreover, the checker might stay in state LIGHT_PENDING without detecting any errors. Since the system specifications usually imply specific requirements to react in a certain time interval,

it is necessary to apply as well the time gesture (see section 2). Thus, before starting the above described specification process, the time gesture is applied to indicate that the following specification mode is time dependant. The effect of the timed specification mode is that when the trigger of a state change is defined, additional transitions and labels are inserted in the checker statechart that check the lower and upper bounds of the time interval.

In the above example, in order to check the upper bound two transitions are added – one at state LIGHT_PENDING and the other one at state DARK_PENDING. The transitions are labelled with a timeout event TM(EN(LIGHT_PENDING),T_2) and TM(EN(DARK_PENDING),T_4), respectively, and a resulting ERROR action.[4] The events are triggered T_n time units after the last entry to state LIGHT_PENDING or DARK_PENDING, respectively. The lower bound check of the state change to LIGHT is realised by the timeout event TM(EN(LIGHT_NOT_YET),T_1) in combination with a new transition between LIGHT_NOT_YET and LIGHT_CHECKED. The latter can only be taken before T_1 time units elapse and therefore results in an ERROR action. The lower bound check of the state change to DARK is specified in a similar way. The concrete timer values T_n cannot be set directly, therefore default values are used that have to be further adjusted (e. g., by the use of a 3D menu in the Virtual Periphery). Note that the lower bound check as well as the upper bound check of the time interval can be omitted.

The resulting statecharts are shown in figure 5. In addition to the above mentioned statecharts for interface objects, another statechart is generated during the specification process. While interacting with the Virtual Periphery, the user triggers certain input interface objects (e. g., a switch) to change state. These user statecharts cannot be defined as behaviour templates in a library, since their states and transitions depend entirely on the test specification for the SUT.

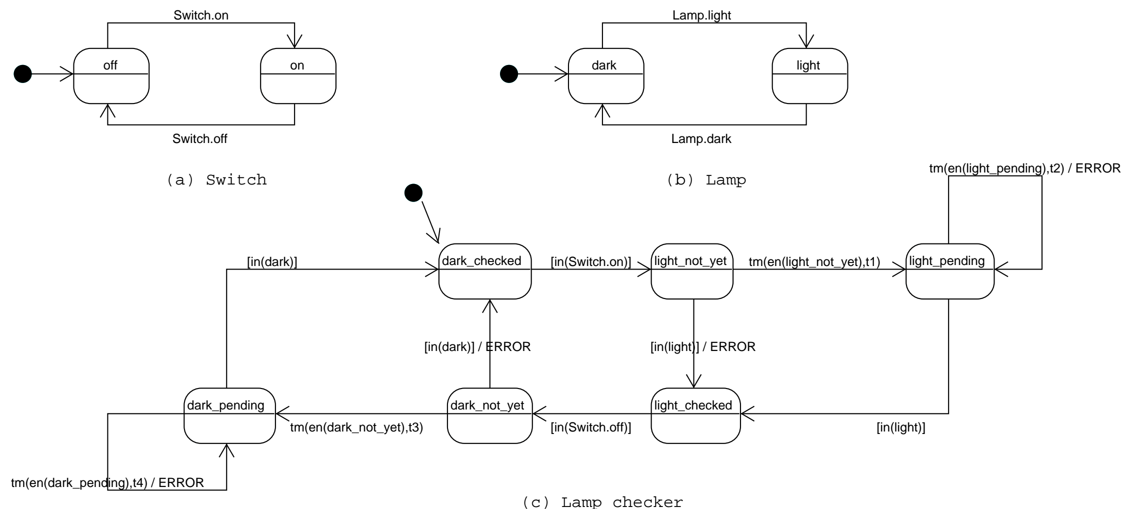


Figure 5: Instantiated and extended statecharts

[4]A more detailed error handling is necessary but is not discussed in this paper.

4 Testing with Statecharts

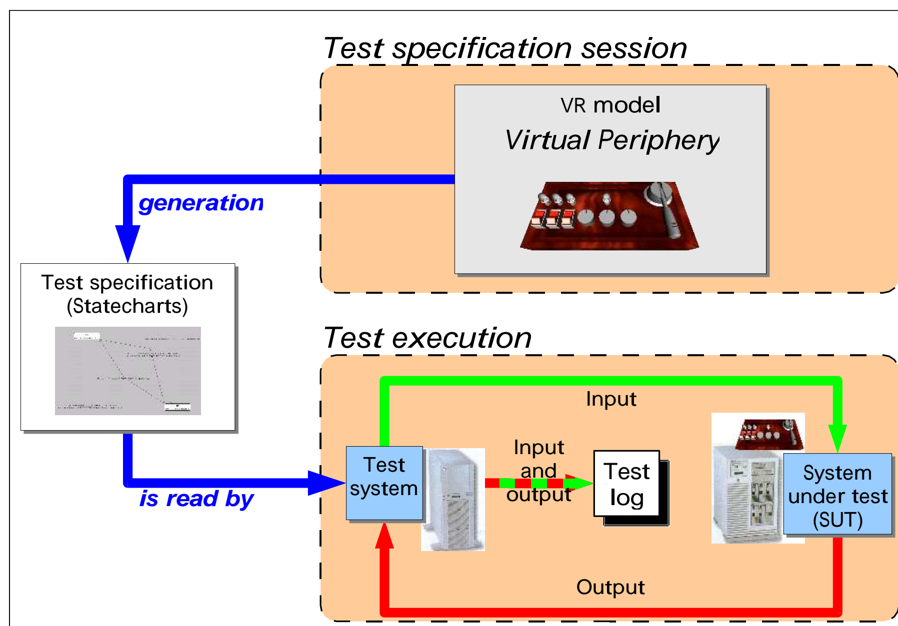


Figure 6: Test process overview

An overview about the complete test process is given in figure 6. While, on the one hand, the statecharts (as discussed in Section 3) are the result of a test specification session, they are, on the other hand, the input to the test system. We focus in our work on RT Tester. RT Tester generates and executes tests automatically by sending inputs to the SUT. The tests are evaluated on the fly based on the given inputs and the outputs coming from the SUT. See [Pel02] and [PT02] for application examples and [Pel98] for the theoretical background.

Since the RT Tester tool accepts test specifications as *labelled transition systems (LTS)* to allow the use of arbitrary formal test specification languages that can be translated into an LTS, we have to provide such a translation relation.

This translation depends on the statechart semantics used during the specification process. Different semantics are available for statecharts: Harel introduced statecharts in 1987 (see [Har87]) and gave a formal semantics in [HPSS87]. A variant described in [HN96] has been implemented in the STATEMATE tool[5]. Statecharts have as well been integrated in UML (see [Obj]) with a slightly different semantics. Other semantics have been discussed as well, and a comparision of different statechart semantics is given in [vdB94]. As well, different approaches for the translation of statecharts into LTS have been discussed, see e. g. [US94], [Lev96] and [Joh99]. Most variants are tailored to meet specific needs. We need a semantics that can at least deal with our timing constraints and which has a step semantics with a greedy approach. Nevertheless, we are currently investigating the specific needs of our approach with respect to the statechart semantics. Thus, we can yet neither provide

[5]STATEMATE is a commercial tool by i-logix (http://www.ilogix.com) and is actually applied to industrial projects.

a complete algorithm for the translation nor a precise semantics for the statecharts used in our approach.

5 Virtual Periphery Generation and Reuse

One crucial point within our approach is the creation of the Virtual Periphery itself. Since the manual generation of the Virtual Periphery is a very time consuming and thus expensive task and moreover highly SUT dependant, it is desirable to minimise any manual effort to create the Virtual Periphery. Hence, we want to discuss in the following an approach to support the generation of the Virtual Periphery. Figure 7 gives a first overview.

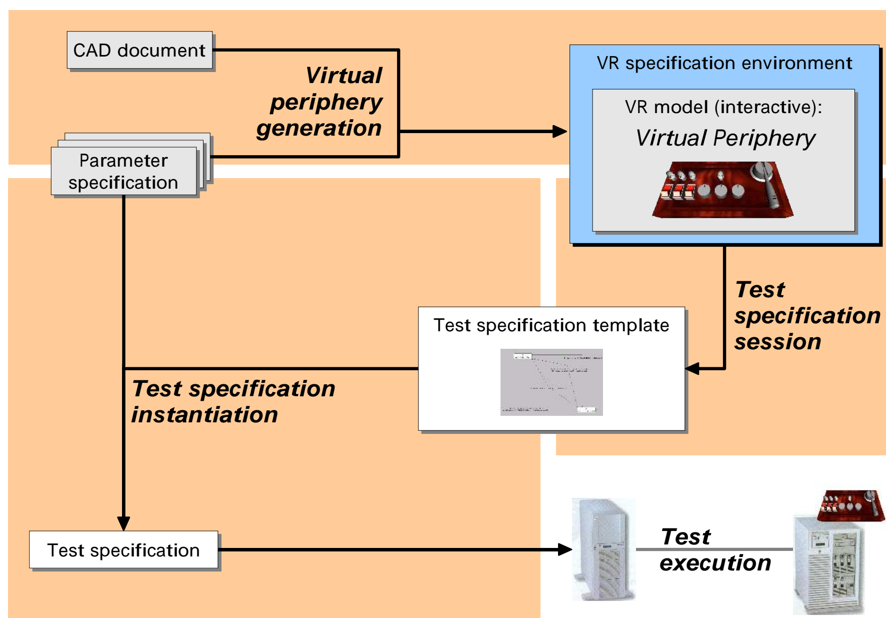


Figure 7: Virtual Periphery generation and test process

In most application areas, there are static geometry models of the SUT's environment and/or the SUT itself. This collection of documents – typically CAD documents – can be converted straight-forward into an appropriate virtual reality model.

Additionally, some SUTs are equipped with some kind of *parameterisation module* in order to configure system features. Consider the *Cabin Intercommunication Data System* that is used within Airbus airplanes. It contains the so-called *Cabin Assignment Module* that parameterises for example, how many attendant handsets are available in the cabin and to which controllers they are connected. A similar example is the number and mapping of passenger service units[6] to seat rows and aisles. Although the parameterisation modules are typically domain specific, once an evaluation is realised it can be used to generate appropriate variants of the Virtual Periphery depending on the specific parameter values.

[6]A passenger service unit is a collection of signs, keys, lamps, . . . above the passenger's seat.

Typically, only an intermediate format – a non-interactive virtual reality model – can be derived automatically from the given documents. Hence, the interface objects have to be inserted or replaced manually by interactive objects in order to specify the tests as discussed in the previous sections. This process can be supported by libraries of interface objects which contain for each type of interface object its geometry as well as the corresponding behaviour template (see sections 2, 3).

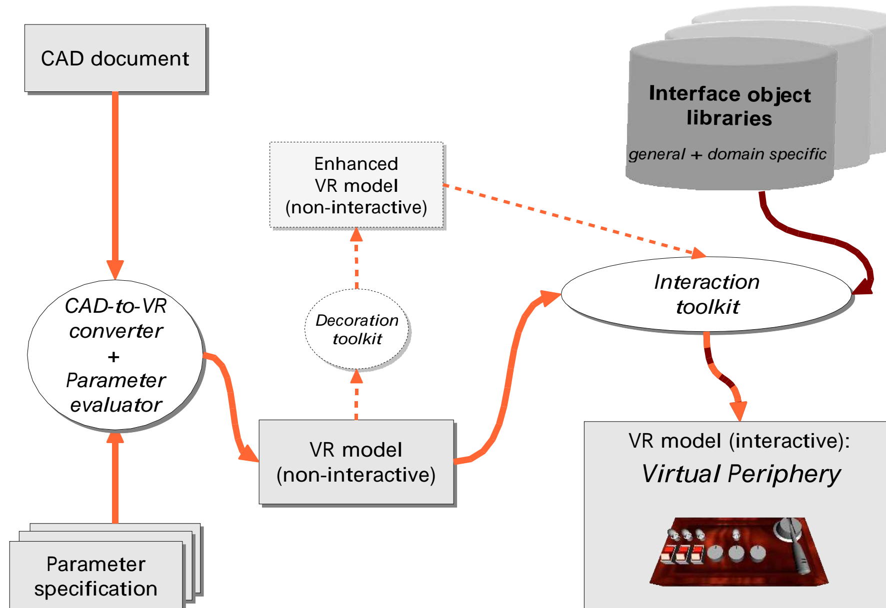


Figure 8: Detailed Virtual Periphery generation

Figure 8 gives a detailed view of the Virtual Periphery generation. It also contains an optional step via an enhanced non-interactive model. This may be desired if the Virtual Periphery is supposed to contain decorations like textures, which usually are not provided within the CAD documents. Enhancing the model is a manual activity similar to the insertion of interface objects.

While the previously described generation approach itself reuses the CAD documents and the parameterisation module (i. e., documents generated not specifically for the purpose of testing), it is as well possible to reuse parts of the test specification fragments based on concrete parameter values. Considering, for example, the reading lights in an airplane, all reading lights can be switched on or off by a central button (using the selection mechanism described in Section 2 to select all reading lights). Additionally, each reading light can be switched on or off by a toggle switch above the passenger seat. Nevertheless, it is neither desirable to specifiy this specification part for each reading light separately nor should it be necessary to specify the behaviour of all reading lights once again, if the parameter value denoting the number of reading lights has changed. In contrast, a test specification template could be used which is instantiated with a concrete set of parameter values before testing (i. e., more precisley before generating the LTS). This approach is visualised in the lower left part of Figure 7 focussing on the test specification instantiation based on the generic test specification template and the concrete parameter values defined in the parameterisation module.

6 Conclusion

This article proposed an approach to facilitate the creation of formal test specifications providing interactive virtual reality components. The approach addresses domain experts who are not familiar with formal specification languages and allows them to create basic test specifications quickly and intuitively. Nevertheless, the person interacting with the Virtual Periphery should have a precise understanding of concepts like parallelism and the sequencing of events.

Thus we expect the benefit of our approach to be:

Simple specifications Simple specifications can be developed without applying elaborate concepts and thus without a detailed understanding of the underlying concepts of the formal specification language.

Team development Within a test team, domain experts and test experts can cooperate to extend the simple specifications.

Introduction to formal specification languages Additionally, our approach can be used to become familiar with formal specification languages in an intuitive way and thus to gain necessary expertise in it. Eventually, the expert will then even prefer to define the test specifications using directly the formal specification language.

The Virtual Periphery described in this article is partially implemented using Java3D which is an API for the general purpose, object oriented programming language Java. It is proved to be superior to VRML which we used during earlier efforts (see [BF99], [PBFE99]), because Java3D allows more flexible modelling. For further information concerning Java, Java3D and VRML see [GJS97], [Jav00] and [VRM97].

One main design guideline during the implementation is the use of conventional personal computers without extraordinary input or output devices. Hence, all interaction must be possible by mouse, keyboard and low cost microphone. Our 3D cursor is carefully designed to be used with the mouse to allow movements and rotations to be gained from two-dimensional mouse movements combined with so-called modifier keys (e. g., pressing of mouse buttons). Furthermore, the output has to be appropriate for conventional monitor screens and stereo pairs of speakers. Note that this is the reason why no haptical output like force feedback is available with the Virtual Periphery.

Nevertheless, it is possible to enhance the virtual reality feeling with special equipment. There is no inherent restriction of our 3D cursor to be used with the mouse, so that alternatively a data glove could be used. For visual output, a head mounted display as well as stereoscopic viewing solutions can be applied. As virtual reality audio enhancement, dolby surround or similar techniques are available. To gain an overview about virtual reality equipment refer for example to [MG96].

Future work will include the definition of a formal statecharts semantics and a corresponding translation to LTS that can be used by the RT Tester tool. Since the

formal language is exchangeable as far as there is an appropriate LTS representation, further evaluation of appropriate formal specification languages is planned. In particular, we will consider hybrid automata (see [Hen96]) that enable modelling of continuous behaviour and are in this respect more expressive than statecharts. However, the chosen language has essential impact on the interaction's semantics within the virtual reality.

Another point that is subject to further research concerns the test evaluation that is so far based on the generated test specification. A way to map an event from the test log of a test run to the corresponding Virtual Periphery representation would be valueable in order to interpret errors and warnings or even to find inconsistencies within the test specification itself. Since the test execution is based on labelled transition systems, this is not a trivial task. Even the mapping to the corresponding part of the statechart is non-trivial.

Finally note that although we focus on developing test specifications, our approach is not restricted to this kind of specifications but can be expanded to more general specifications.

References

[BF99] Stefan Bisanz and Ingo Fiß. Grafischer Entwurf von CSP-Spezifikationen für den Test eingebetteter Echtzeitsysteme. Master's thesis, Universität Bremen, February 1999.

[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.

[GJS97] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* The Java Series. Addison-Wesley, 1997. http://java.sun.com/docs/books/jls/html/.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[Hen96] Thomas A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science (LICS)*, pages 278–292. IEEE Computer Society Press, 1996.

[HHN86] E. L. Hutchins, J. D. Hollan, and D. A. Norman. Direct manipulation interfaces. In D. A. Norman and S. W. Draper, editors, *User Centered System Design: New Perspectives on Human-Computer Interaction*, pages 87–124. Erlbaum, Hillsdale, NJ, 1986.

[HN96] David Harel and Amnon Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[HPSS87] D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman. On the formal semantics of statecharts. In *Proceedings, Symposium on Logic in Computer Science*, pages 54–64. The Computer Society of the IEEE, June 1987. Extended abstract.

[Jav00] The Java 3D^{TM} API Specification. Version 1.2. http://java.sun.com/products/java-media/3D/index.html, April 2000.

[Joh99] Sebastian John. Zur kompositionalen Semantik von objektorientierten Statecharts. Master's thesis, Technische Universität Berlin, August 1999.

[Lev96] Francesca Levi. A process language for statecharts. In *Logical and Operational Methods in the Analysis of Programs and Systems*, pages 388–403, 1996.

[MG96] Tomasz Mazuryk and Michael Gervautz. Virtual reality - history, applications, technology and future. Technical Report TR-186-2-96-06, Vienna University of Technology, Institute of Computer Graphics and Algorithms, A-1040 Wien, February 1996. Available as http://www.cg.tuwien.ac.at/research/TR/96/TR-186-2-96-06Abstract.html.

[Obj] Object Management Group (OMG). *OMG Unified Modeling Language, Specification*. Available at http://www.omg.org/uml/.

[PBFE99] J. Peleska, S. Bisanz, I. Fiß, and M. Endreß. Non-Standard Graphical Simulation Techniques for Test Specification Development. In H. Szczerbicka, editor, *Modelling and simulation: A tool for the next millenium. 13th European Simulation Multiconference 1999*, volume 1, pages 575–580, Delft, 1999. Society for Computer Simulation International.

[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.

[Pel02] Jan Peleska. Formal methods for test automation - hard real-time testing of controllers for the airbus aircraft family. In *Proc. of the Sixth Biennial World Conference on Integrated Design & Process Technology (IDPT2002), Pasadena, California*. Society for Design and Process Science, June 2002. To appear.

[PT02] J. Peleska 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.

[US94] Andrew C. Uselton and Scott A. Smolka. A compositional semantics for statecharts using labeled transition systems. In *International Conference on Concurrency Theory*, pages 2–17, 1994.

[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.

[VRM97] VRML Consortium. The Virtual Reality Modeling Language. http://www.web3d.org, 1997. International Standard ISO/IEC 14772-1:1997, Part 1 – 3.