Model-based Testing of a Reactive System with Coloured Petri Nets

Simon Tjell (tjell@daimi.au.dk)
Dep. of Computer Science, University of Aarhus, Denmark

Abstract: In this paper, a reactive and nondeterministic system is tested. This is done by applying a generic model that has been specified as a configurable Coloured Petri Net. In this way, model-based testing is possible for a wide class of reactive system at the level of discrete events. Concurrently executed tasks are specified at a high level of abstraction and test traces are collected through state space analysis of the model.

1 Introduction

This paper describes a generalized model for simulating and generating discrete event test traces for a specific group of embedded system. This group is defined by a set of properties: The system consists of one or more CPUs, each execution a set of tasks with prioritized preemption. The CPUs are connected by a communication bus through which the tasks are able to exchange messages - this is the only means of synchronization between the CPUs. Apart from exchanging messages with other tasks, each task is able to react on and actively produce events. These events are exchanged with the environment.

In the CPN-model [Jen92], a system belonging to this group can be specified at a relatively high level of abstraction and state space analysis is then applied in order to extract a set of all possible behavioral traces of discrete events for the given system. These event traces are based on a configurable sequence of input events (stimuli that trig the test output) and are used for comparison with the output from actual implementations of the system being tested. Since the set of traces derive from state space analysis of the CPN-model, it is expected to contain all possible sequences of events that could be expected from the system being specified and modeled - with respect to the level of abstraction being applied in the model. There are basically two reasons for the existence of more that just one possible trace: 1) The combination of different interleaving of tasks in the CPUs due to the multi-threaded operating system being modeled and 2) the observability problem as defined by Fidge [Fid88] (non-causally related events can be observed in arbitrary orderings). In this way, the set of traces capture the nondeterministic behavior of a concurrent system.

The model allows logging of events in both the individual CPUs and in the connection domain - the interface between the system and the physical environments [Jac95].

2 The car-radio case study

A case study describing [WTVL04] a car radio system with navigation functionality has been used throughout the project. This system is concurrent and nondeterministic and suits well for the purpose. The operation of the system can be divided into three basic services:

MMI: The Man-Machine Interface (MMI) service handles interaction with the user through buttons and the display.
Radio: The Radio service provides basic radio functionality as well as the support for receiving traffic announcements through the Traffic Message Channel network [TMC].
Navigation: The Navigation service provides the user with basic GPS navigation. The navigation is integrated with the TMC functions in such a way that traffic announcements can be depicted in a map based on their geographical information.

It is assumed that each service is executed on its own CPU. The services are executed concurrently and the only method for synchronization is through the exchange of messages. These messages are exchanged through a common communication bus that is shared by the CPUs. The case study specifies three applications (use cases) that involve combinations of the services:

Application	**Services**
Change Audio Volume	*MMI* and *Radio*
Address Lookup	*MMI* and *Navigation*
Handle TMC Messages	*MMI*, *Radio* and *Navigation*

Each application involves a collection of tasks distributed within the group of involved services. For example, the MMI service consists of tasks such as *HandleKeyPress*, *UpdateAddress* etc. These tasks are activated when a message is received from another task or when an event such as a keypress occurs. The tasks react by producing a sequence of responses in the form of messages or events. The applications have been specified by simple message-sequence charts, which is sufficient since - in this project - only the order of occurrence of the messages and events is relevant.

3 The CPN-model

The generic model that can be configured to simulate the discrete event behavior of the car-radio case study is developed using Coloured Petri Nets (CPN) [Jen92]. CPN is a formally specified modeling language with a graphical representation. CPN-models specify the structural and behavioral aspects of (distributed) systems in which the state is represented by places holding tokens. *Places* are connected by *transitions* that are able to move tokens between the places of a net and thereby modify the state of the places and the model in general. In this case of modeling a reactive system, the tokens represent entities such as tasks and messages. In the same way, the transitions represent actions such as activation of a task and the execution of an operation.

CPN-models differ from traditional Petri Nets by the fact that the tokens have types and

values - much like in standard programming languages. This makes it possible to distinguish the tokens and simulate tokens moving around between the places.

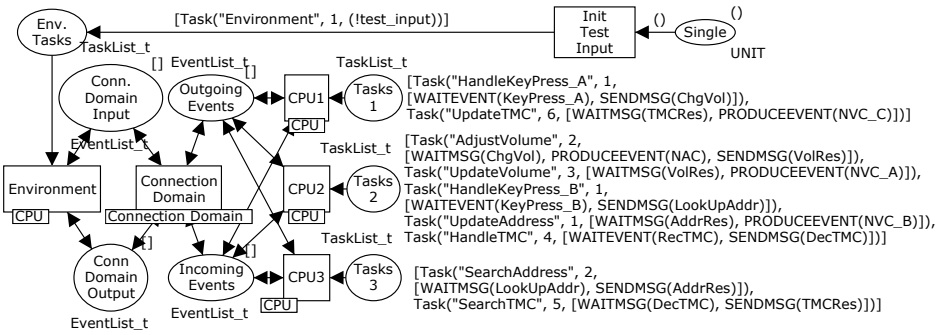

Figure 1: The top-most level of abstraction in the model

This section will give a very brief introduction to CPN along with the description of the model. Please refer to [Jen92] for further details. A CPN-model can be graphically represented as a directed graph that is composed by two basic node-types: Transitions (rectangles) and places (ellipses). These nodes are connected by arcs that indicate the direction in which tokens are moved. An arc either goes from a transition to a place or from a place to a transition. The model caries annotations in CPN ML, which is an inscription language that is based on the Standard ML functional language. The inscriptions are basically used to manipulate and select the tokens that are moved around and they apply to both places, transitions and arcs. A transition typically resembles an event that occurs in the model. When this event occurs, the transition is said to *fire*. When a transition fires, it is able to remove tokens from its input-places (the places that have arcs leading to the transition) and produce new tokens on its output-places - this can intuitively be interpreted as the transition moving around tokens. Remember that a token carries data values (complex or simple). The inscriptions in the arc leading to a transition define the quantity of tokens needed for *enabling* the transition. When a transition is enabled it means that it is ready to fire. Also qualitative constraints can be specified for the tokens from the input-places. This can be done by the use of pattern matching or *guards* in the transition. A guard is a predicate that must evaluate to true for the transition to be enabled. If it is possible to select a set of tokens from the set of input-places while satisfying these constraints for a given transition, that transition is enabled. If another transition depends on some of the same tokens for its enabling, the transitions are said to be in *conflict*. This property is part of what makes (Coloured) Petri Nets a very strong tool for modeling distributed and concurrent systems with shared resources, parallel processes and everything else that comes with this type of systems. The inscription language is also used to specify the initial *marking* of the model - i.e. the initial collections of tokens in the different places of the model.

The CPN modeling language supports the specification of hierarchically structured models. This makes it possible to work with different levels of detail and abstraction as will been seen in the following introduction to the model that has been developed for the work that is described in this paper. Figure 1 shows the top-most *page* of the model. A page

is a part of a model that links to other pages through shared places that cross the levels of hierarchy and thereby enable the exchange of tokens between pages at different layers. Sub-pages are represented in a higher lying page in the form of *substitution transitions*. Examples of this is seen if Figure 1 where certain transitions are labeled with small rectangular labels. These labels specify which sub-pages contains the details for a given transition. A specific label (such as *CPU*) can be used several times in an object-oriented approach to reuse sub-pages to specify the contents of more than one substitution transitions. In this way, the structure and behavior is only specified once and reused to represent - in this case - three CPUs that share a common interface to other pages in the model. Each CPU substitution transition is linked to three types of places in which two are shared with the other CPUs (*Outgoing Events* and *Incoming Events*). The third is individual for the three CPUs (the enumerated *Tasks*-places).

To the right of the three *Tasks*-places, the inscriptions specifying the initial markings for those places are seen. In this case, the tokens on the places are of a complex type namely an ordered collection of another complex type (a record type). The inner record type specifies a single task that is simulated in the CPU. It has a name, a priority, a sequence of steps and an individual program counter. The steps specify the operations of a task at a very high level of abstraction. So far, a small expandable set of operations has been specified: *WAITMSG*, *WAITEVENT*, *PRODUCEEVENT*, *SENDMSG*, *RESTART*, *TERMINATE*. All the event/message related operations take a parameter that specifies which event or message to produce or wait for. This set of operations have been sufficient for modeling the car-radio case-study.

The overall structure of this top-level page shows how the model consists of three CPUs that are connected to the environment through a connection domain. A peculiar detail is the fact that the environment itself is also modeled by use of the CPU-page and in this way its behavior is specified using the same set of operations. It only differs from the regular CPUs by being connected oppositely to the connection domain in order to make its outgoing events become incoming events for the CPUs - and vice versa.

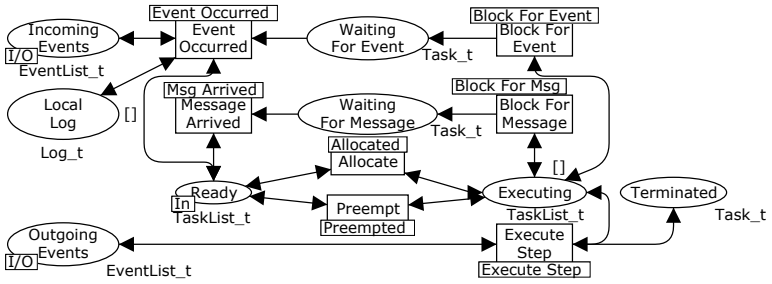

Figure 2: A more detailed view of the behavioral specification of a CPU in the model

Next, we look into a deeper level of detail; the sub-pages specifying the details of a CPU. This is seen in Figure 2. Here, it is seen that this sub-page links to even lower levels of details by having substitution transitions itself. The CPU-page is linked to the top-level page through *port* places. An example of this is the *Ready*-place that is annotated with

a small *In* label indicating the direction of the link. This link ensures that the port place and the place to which it is linked (the *socket* place) always share the same marking. In the specific example, the *Ready*-place is linked the *Tasks*-place of the CPU and thereby contains the task tokens that were just described. Most of the places in this page (*Ready*, *Executing*, *Terminated*, *Waiting for Message* and *Waiting for Event*) represent different states that a task can be in. This state will change as events occur, messages are received and operations are executed. Since all places can hold a set of tokens, the CPU-page is able to simulate the multi-threaded execution of a set of tasks. The model of the operating system has been based on a layout found in [Gom00], which is used for a referential structure of an operating system at a high level of abstraction. It is seen how all transitions in this page are substitution transitions of which the details are specified in sub-pages. An example of this is the *Preempted*-transition. In this page it is merely shown to have connections to the *Ready* and the *Executing*-places. In the sub-page (not shown), the behavior for priority-based preemption is specified. When tasks are ready, the task with the higher priority will be allocated the CPU and be moved to the *Executing*-place. When the task is in that place, its operations will be executed in the specified order along with the program counter being incremented - this is done by the *Execute Step*-transition. A task can wait for the arrival of a message or an event. When a task starts waiting it is moved away from the *Executing*-place and the CPU can be allocated to another task. In this way, tasks are executing until they are either preempted (by a task with higher priority), terminated or start waiting for an event or a message. When the operations are executed, events can be produced and messages can be sent. The latter is done through a bus that connects all the CPUs. The bus is specified by a *fusion set*, which is a simple way to link together places across the model.

3.1 Conformance Checking

This section describes how the model is used to test the conformance of the output from an actual implementation. For this purpose, a simple multi-threaded implementation of the car-radio case study has been implemented in Java. This test implementation is stimulated with a sequence of input events and responds by producing a sequence of output events. The combined sequence of input and output events is recorded and stored as a log file while executing the test implementation. All the tasks in the case study are implemented as independent threads. These threads communicate asynchronously by exchanging messages through a dispatcher. This allows the threads to interleave in many combinations - and this again reflects the nondeterministic nature of the concurrent system in the event traces that are generated and recorded.

In order to collect event traces from many executions, another simple program has been developed. This program executes the test implementation a specific number of times while recording its output and appending this output to a combined log file. While this is done, the output is automatically translated into a matching CPN ML multiset (a set that can contain more than one element with the same value). This *multiset* consists of a number of event traces - one from each execution of the test implementation. Since it is

translated into CPN ML, it can be used by of the conformance checking method.

When a large number of event traces has been collected from the implementation, these traces will be checked individually to determine whether they conform with one of the expected event traces - based on the knowledge from the model.

3.2 State Space Analysis for Generating Output Traces

This subsection describes how the referential event traces to which the output from the implementation are generated by use of state space analysis of the model.

The state space of a CPN-model can be calculated (fully or partially) by use of a computer based tool. In this project, CPN Tools [CPN] has been used for both specification and state space analysis of the model. In the context of CPN-models, the state space is a directed graph in which the nodes represent all possible markings - i.e. states - of the model (if a full state space is calculated). The arcs leading from one state to another represent the firing of transitions with all possible token values (*bindings*).

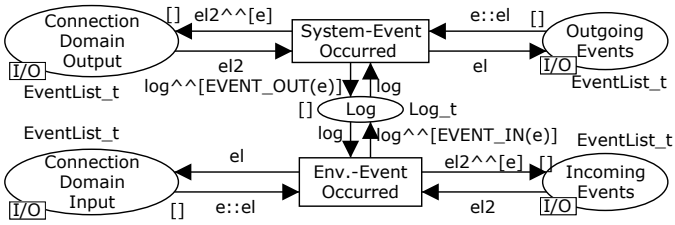

Figure 3: The Connection Domain

Figure 3 shows the details of the *Connection Domain*-page. The *Log*-place is used for logging all events that cross the connection domain. The events are appended to an ordered collection as they are observed. Remember that the simulation of the model resembles the nondeterministic behavior of the concurrent system being modeled. This property will be reflected by the fact that the ordering in the list in the *Log*-place (potentially) differs between repeated executions of the model. The contents also depend on the input events with which the modeled system is triggered. Given a specific sequence of input events, the state space will reflect all expected orderings of input and output events in the *Log*-place as they are observed in the connection domain. By use of tool-based analysis of the *multiset bounding properties* of the state space, all possible markings of a given place can be collected. This is done with respect to the *Log*-place for a set of different sequences of triggering input events. This set is composed of all possible permutations of sequences containing the three input events (*KeyPress_A*, *KeyPress_B* and *ReceiveTMC*. This results in six different sequences which again results in the generation of six different state spaces for the model. All possible orderings of the observed events are collected in a single function in CPN ML (*model_output()*). This function returns a multiset containing all sequences of events that could be expected from stimulating the implementation with the

six permutations. Likewise, the implementation is stimulated randomly with one of the six permutations every time an execution is started.

What the state space analysis does is basically selecting a subset of all possible event traces. When the model is configured to simulate the car-radio case study, this results in 205 different traces. This number should be compared to the number of possible permutations of 7 input and output events that are observed in one trace - this number is 5040 different traces (7!). The number of different traces from the model depends heavily on the level of nondeterminism of the system being modeled.

3.3 A method for Checking the Output from an Implementation

When the output from the model has been collected by using state space analysis along and a collection of observed event traces from the implementation has been recorded, it is time to check the conformance of the implementation. This is done by applying a quite simple algorithm. The CPN ML specification is shown here:

```
fun LoadImplementationOutput() = let
 val fid = TextIO.openIn("batch_output.sml");
 val log = Log_t.input_ms(fid)
in TextIO.closeIn(fid); ms_to_list(log) end;

fun CheckImplementationOutput(log::loglist) =  (cf(log, model_output()) > 0)
 :: CheckImplementationOutput(loglist) | CheckImplementationOutput(nil)  = empty;

CheckImplementationOutput(LoadImplementationOutput());
```

The algorithm reads the entire collection of event traces from the implementation executions and then checks them one by one. The check of an individual trace is performed by checking whether or not that trace exists in the collection of event traces from the model. If the event trace from the implementation is found in the output from the model, it is approved. The result of conducting the entire conformance check is a multiset of booleans in which the number of true values describes the number of event traces from the implementation being approved.

The conformance checking algorithm has been applied with the test implementation - and this has helped to check both the algorithm and the test implementation. The test implementation was successfully tested with batch executions in the order of some thousands that were tested with the conformance checking algorithm - and passed.

4 Related Work and Conclusion

This paper mainly relates to work where formal methods have been applied to specify the expected behavior of an implementation and then used for testing the actual behavior of the implementation. This approach has previously been applied for testing in a wide range of domains Smart Cards [PP05], standard conformance (Java and POSIX) [FHP02],

desktop applications [RR00], embedded automotive systems [CFS05], processor verification [AGL+95], real-time systems [MLN03] and concurrent systems [Tre99]. Similarly, a wide range of formal methods have been chosen to specify the models. A presentation of sane choices of modeling techniques for specific applications is found in [EF01].

A model-based framework for simulating and conformance checking reactive systems based on a high-level specification of their discrete event behavior has been presented. The approach has been successfully demonstrated with a real-life case study. One of the future plans for this project is to evolve the way the tasks are specified in the form of high-level operations, while still maintaining that high level of abstraction. This will widen the range of systems that can be simulated.

References

[AGL+95] Aharon, Goodman, Levinger, Lichtenstein, Malka, Metzger, Molcho, and Shurek. Test Program Generation for Functional Verification of PowerPC Processors in IBM. In *DAC*, 1995.

[CFS05] Mirko Conrad, Ines Fey, and Sadegh Sadeghipour. Systematic Model-Based Testing of Embedded Automotive Software. *Electr. Notes Theor. Comput. Sci.*, 111:13–26, 2005.

[CPN] CPN Tools. http://wiki.daimi.au.dk/cpntools/cpntools.wiki.

[EF01] El-Far. Enjoying the Perks of Model-Based Testing. In *STARWEST 2001*, 2001.

[FHP02] Eitan Farchi, Alan Hartman, and Shlomit S. Pinter. Using a model-based test generator to test for standard conformance. *IBM Systems Journal*, 41(1):89–110, 2002.

[Fid88] Colin J. Fidge. Time stamps in message-passing systems that preserve the partial ordering. *Australian Computer Science Communications*, 10(1):56–66, 1988.

[Gom00] Hassan Gomaa. *Designing Concurrent, Distributed, and Real-Time Applications with Uml*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[Jac95] Jackson. *Software Requirements And Specifications*. Addison-Wesley Pro., 1995.

[Jen92] Kurt Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1992.

[MLN03] Mikucionis, Larsen, and Nielsen. Online On-the-Fly Testing of Real-time Systems. Technical Report RS-03-49, 2003.

[PP05] Prenninger and Pretschner. Abstractions for Model-Based Testing. *ENCS*, 116, 2005.

[RR00] Rosaria and Robinson. Applying models in your testing process. *Information & Software Technology*, 42(12), 2000.

[TMC] What is RDS-TMC? http://www.tmcforum.com/en/about_tmc/what_is_tmc/what_is_tmc.htm.

[Tre99] Tretmans. Testing Concurrent Systems: A Formal Approach. In *CONCUR*, 1999.

[WTVL04] E. Wandeler, L. Thiele, M. Verhoef, and P. Lieverse. System architecture evaluation using modular performance analysis - a case study. In *ISoLA*, 2004.