Two Interfaces to the CPN Tools Simulator

Michael Westergaard and Lars Michael Kristensen

Department of Computer Science, University of Aarhus,
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark,
Email: {mw,kris}@cs.au.dk

Abstract. Coloured Petri nets (CP-nets or CPNs) is a useful modeling formalism for formally describing concurrent systems, and CPN Tools provides a mature environment for constructing, simulating, and performing simple analysis of CPN models. Sometimes, this does not suffice, however. For example, if one wishes to extend the analysis capabilities or to integrate CPN models into other programs. In this paper we present two new interfaces which facilitate this. One is written in Standard ML and is very close to the simulator component of CPN Tools, providing a solid foundation for developing advanced analysis tools. The other interface is written in Java and provides an object-oriented representation of CPN models as well as a means to load models created using CPN Tools. Furthermore, the Java interface provides a high-level interface to the simulator component facilitating integration of simulation of CPN models into other programs. We illustrate the interfaces by providing the complete implementation of a command-line state space exploration tool. The interfaces are available to interested parties.

1 Introduction

Coloured Petri nets (CP-nets or CPNs) provide a useful modeling formalism for formally describing concurrent systems, such as network protocols [12] or work-flows in companies [8]. CPN Tools [14] provides a mature environment for editing and simulating CPN models, and to a limited degree also for formally verifying that a given model is correct using state space analysis.

Sometimes, this is not enough, however. The basic problem is that CPN Tools is inherently graphical and cannot be controlled by outside applications. This makes it difficult to use CPN Tools in settings that are outside its scope of interactive use by one user. Such examples include repeated simulation on multiple servers in a grid, which is a useful analysis technique for models that are too large for exhaustive analysis techniques like state space analysis, to describe a complex decision procedure in a parametrised manner for use in a regular application, and allowing users to set parameters of a model using a custom user interface and just present the end-result of a simulation. It is also difficult to implement new analysis techniques such as new more efficient state space methods or completely different analysis methods (e.g., coverability graphs, bounded model-checking techniques, or invariant analysis), especially if we intend to also build a user-friendly interface for the new methods.

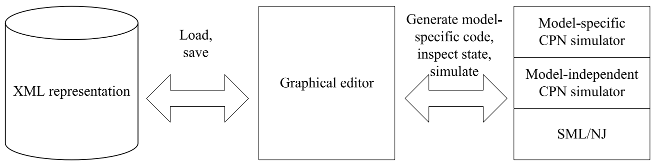
XML representation
Load, save
Graphical editor
Generate model-specific code, inspect state, simulate
Model-specific CPN simulator
Model-independent CPN simulator
SML/NJ

Fig. 1: Architecture of CPN Tools.

CPN Tools basically consists of two components (see Fig. 1), a graphical editor (middle) and a simulator daemon (right). The graphical editor allows the user to interactively construct a CPN model. The model is transmitted to the simulator daemon, which checks it for syntactical errors and generates model-specific code to simulate the model. The graphical editor can invoke the generated simulator code and present the results graphically. The graphical editor can also load and save models using an XML format (left in Fig. 1). The graphical editor imposes most of the previously mentioned restrictions; the simulator daemon is basically a generic Standard ML/New Jersey (SML/NJ or SML) [15] run-time environment and compiler with functions for syntax checking CPN models. It is obvious that by replacing the graphical editor with our application, we can alleviate most of the limitations imposed by the graphical editor, and this has indeed also been done in different settings [10,16]. The CPN simulator, however, suffers from two problems making such a replacement difficult. Firstly, the protocol used for communication between the graphical editor and the simulator is rather low-level and complex to implement. Secondly, the CPN simulator is optimised for simulation and incremental code generation making it difficult to use for other purposes as the model-specific code is difficult to use.

In this paper we propose two new interfaces to the CPN simulator[1]. Neither aim to replace CPN Tools as editor for CPN models, but rather to allow people to make experiments with the formalism. Both of the interfaces have been developed as part of the ASCoVeCo [1] project and the ASAP model checking platform [11], but are believed to be useful in other settings as well. Neither of the interfaces are intended for end-users; both of the interfaces provide rather low-level simulation primitives, which can be used by programmers to build new generic tools. We present the interfaces in the context of formal verification because that has been our motivation for developing the interfaces, but numerous applications can build upon the foundation to allow more high-level use of the CPN simulator. One of the interfaces is written in Java and the other in SML. In Fig. 2 we see how the new interfaces augment and replace parts of CPN Tools. The Java interface (middle) consists of an object-oriented representation of a CPN model, the ability to transmit this representation to the simulator and to programmatically perform simulation and inspection of the current state in the simulator. Furthermore, it includes an importer module which can import models created using CPN Tools. In effect, this allows programmers to load a model created using CPN Tools (left), instantiate a simulator for this model and perform simulation of the

[1] The interfaces are available to interested parties; send an email to `ascoveco@cs.au.dk` for more information.

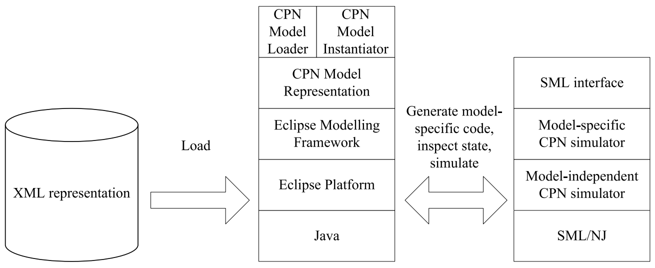
XML representation
Load
CPN Model Loader
CPN Model Instantiator
CPN Model Representation
Eclipse Modelling Framework
Eclipse Platform
Java
Generate model-specific code, inspect state, simulate
SML interface
Model-specific CPN simulator
Model-independent CPN simulator
SML/NJ

Fig. 2: Architecture of new interfaces.

model in their own applications, which can be anything from a simple command-line utility to a full-fledged CPN editor. The SML interface (right in Fig. 2) encapsulates the complex data-structures used in the simulator, and provides a simple frozen interface to the state of a CPN model which facilitates very fast simulation. This is in particular useful for efficient analysis, e.g., by means of state spaces, but applicable for any application that requires fast execution of transitions with little or no user-interaction.

The rest of this paper is structured as follows: In the next section, we introduce a simple example, that is used throughout this paper. In Sect. 3, we describe the SML interface to the simulator, and in Sect. 4, we describe the Java interface to the simulator. These two sections are independent of each other. In Sect. 5 we use the two interfaces to create a simple command-line tool for state space analysis of CPN models. Finally, in Sect. 6, we sum up our conclusions and provide directions for future work.

2 Example CPN Model

Throughout this paper we will use a CPN model of a simple stop-and-wait protocol with one sender and two receivers. The top module of the model can be seen in Fig. 3, where we have a substitution transition for the sender, the network, and one for each receiver. The network has a maximum capacity modeled by the Limit place. If the network still has available capacity, the sender (Fig. 4 (left)) transmits packets onto the A place. The place Send contains the packets to send. The network (Fig. 4 (middle)) then transmits the packet to B1 and B2, optionally dropping one or both of the packets. The receivers (Fig. 4 (right)) receive the packets on Received and transmit back acknowledgements onto C1 or C2, which the network transmits to D, optionally dropping one or both. When the sender receives acknowledgements from both receives, the NextSend counter is updated and the cycle restarts. We observe that the model consists of four different modules: Top, Sender, Network, and Receiver. The Receiver module is instantiated twice as Receiver 1 and Receiver 2 in the module Top.

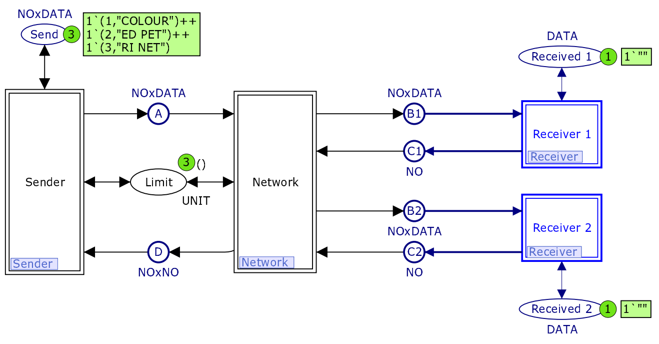

Fig. 3: Top page of a simple stop-and-wait protocol model with two receivers.

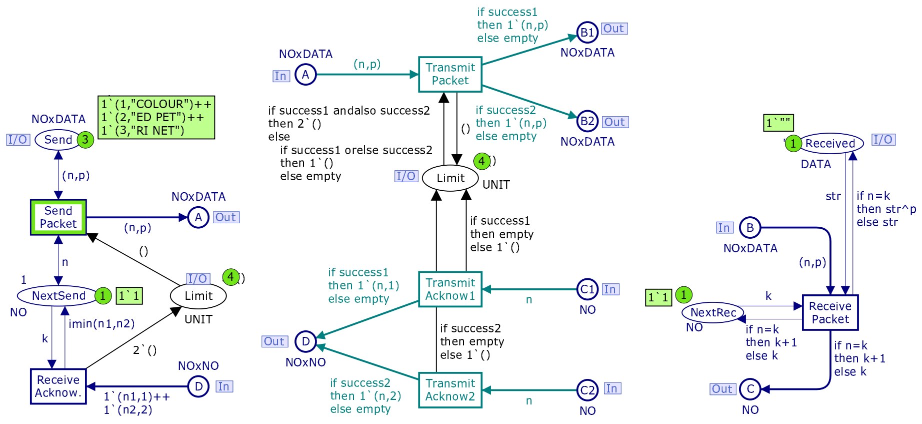

Fig. 4: Sender (left), network (middle), and receiver (right) modules of the protocol.

3 The SML CPN Model Interface

In this section we present the old SML interface to the simulator and some of its shortcomings. We also present our new interface and explain why it is superior. The aim of the SML model interface is to provide efficient access to the CPN simulator, in particular with the purpose of implementing efficient analysis methods. To support this, the SML interface provides an interface to the state of a CPN model and to execute enabled transitions. For performance reasons, this interface is written in the same language as the CPN simulator itself, namely SML/NJ [15]. We suggest that all applications that are algorithmic in nature use the SML interface described in this section. Using SML as implementation may seem a bit strange as it is not as well-known as, e.g., Java. The choice makes sense, however, both because this interface is the fastest as it is written in the same language as the simulator itself and because SML is a useful language for declaratively implementing complex algorithms due to its functional paradigm.

3.1 The Old SML Interface

In Listing 1.1 we see part of the current interface for the model in Figs. 3 and 4. In lines 1–10 we see the definition of the place NextRec in the module Receiver. We first notice that the relationship to the place and module is not immediately visible, as the place is only referred to by a generated identifier (CPN'placeid168). All places reside at the top level, so the modularity of the model is not visible in the interface. The functions get and set (ll. 7–8) take as parameter an instance number, which is the internal number of the instance of the place. This number is not immediately derivable from the model (we have, e.g., no guarantee that the instance corresponding to Receiver 1 has number 1). The ims.cs ms type is a multi-set over the type of the place, in this case NO.

The rest of Listing 1.1 shows representations of three different transitions, Send Packet from Sender (ll. 11–15), Transmit Acknow1 from Network (ll. 16–21), and Transmit Packet from Network (ll. 22–29). Like places, all transitions are referred to by a generated identifier rather that their user-recognisable name. Transitions, like places, live at the top-level, and the CPN'occfuns (ll. 12, 17–18, and 23–25) take an internal instance number as the first parameter. The last parameter given to CPN'occfun is a boolean indicating whether the step-counter should be incremented. This is used internally by the simulator for handling monitors, and during normal simulation should always be set to true. The middle parameter to a CPN'occfun describes the binding of the variables of the transition. For Send Packet, this consists of a record containing all variables. The two

Listing 1.1: Current interface.

```
 1  structure CPN'placeid168: sig
 2    structure ims: sig
 3      structure cs: COLORSET
 4      type cs = cs.cs
 5      ... (* 1 type definition and 22 functions *)
 6    end
 7    val get: int -> ims.cs ms
 8    val set: int -> ims.cs ms -> unit
 9    ... (* 2 constants and 8 functions *)
10  end
11  structure CPN'transitionID1264271480: sig
12    val CPN'occfun: int * {n:NO, p:DATA} * bool -> CPN'Sim.result * string list
13    val CPN'bindings: int -> {n:NO, p:DATA} list
14    ... (* 5 constants, 3 variables, and 6 functions *)
15  end
16  structure CPN'transitionID1264276591: sig
17    val CPN'occfun:
18      int * ({n:NO} * BOOL) * bool -> CPN'Sim.result * string list
19    val CPN'bindings: int -> ({n:NO} * BOOL) list
20    ... (* 5 constants, 3 variables, and 6 functions *)
21  end
22  structure CPN'transitionID1264276586: sig
23    val CPN'occfun:
24      int * ({n:NO, p:DATA} * {success1:BOOL, success2:BOOL}) * bool
25      -> CPN'Sim.result * string list
26    val CPN'bindings:
27      int -> ({n:NO, p:DATA} * {success1:BOOL, success2:BOOL}) list
28    ... (* 5 constants, 3 variables, and 6 functions *)
29  end
```

transmit transitions are more complex. The technical reason is that, in the case of the Transmit Acknow1, the variables n and success1 are not correlated in any way, and can be bound independently, so by separating them it is possible to find legal bindings for the transition more efficiently. The CPN'occfun for Transmit Packet is just a more complex example of this. The result of CPN'occfun is a result from the simulator, indicating whether the transition was successfully executed, whether the transition was disabled, or whether the transition was not enabled a the current time stamp (for timed models). Additionally, a list of descriptive error messages may be returned. All transitions also have a function, CPN'bindings (ll. 13, 19, and 26–27), which given an instance number returns a list of all enabled bindings using the same grouping of variables as CPN'occfun.

This interface is well-suited for high-performance simulation and incremental code generation. By distributing the state to multiple structures, it is possible to update only markings of places affected by the execution of a given binding element (transition with associated binding of all variables), making the execution independent of the size of model. This also makes the enabling calculation more efficient, as the enabling is only affected for transitions connected to modified places (and we can even exploit monotonicity of enabling to further improve the enabling calculation). Furthermore, as all places and transitions are represented as separate structures, incremental code generation is independent of the size of the model. Adding a place or transition simply means we have to add a new structure. Modifying a transition only requires the regeneration of a single structure, and modifying a place only requires that we regenerate the structure corresponding to the place and all structures corresponding to transitions connected to the place, which is in practise a low number. Finally, during simulation, we are just interested in whether a transition is enabled, and, if so, to execute one enabled binding element. This is greatly facilitated by grouping the variables of transitions, as there is no reason to calculate all binding elements, which can be found as elements of the Cartesian product of elements of each group.

The properties of the interface facilitate an editor with incremental syntax check and efficient simulation of CPN models, but the requirements for a state space tool are different as we are dealing with many states (as opposed to just one during simulation), requiring that it is possible to represent more than one state. Also, we need to obtain all enabled binding elements in a given state. As the state is distributed across multiple structures in the old interface, it is difficult to represent more than one state at a time, as we would need to traverse all structures to read the marking of each place. As the enabling calculation of transitions is distributed across many structures, gathering all enabled transitions requires checking enabledness of transitions individually. Finally, the old interface is not very user-friendly, as we refer to all nodes using internal generated names and instance numbers not easily obtainable by the user.

3.2 The New SML Interface

Instead, we define a completely new interface to CPN models. The interface is designed with state space analysis in mind, but can of course be used for other

Listing 1.2: Model interface.

```
 1  signature MODEL = sig
 2    eqtype state
 3    eqtype event

 5    exception EventNotEnabled

 7    (* Get the initial states and enabled events in each state            *)
 8    val getInitialStates: unit -> (state * event list) list

10    (* Get the successor states and enabled events in each successor state  *)
11    val nextStates: state * event -> (state * event list) list

13    (* Execute event sequence, return resulting states and enabled events   *)
14    val executeSequence: state * event list -> (state * event list) list

16    (* String representations of states and events                        *)
17    val stateToString: state -> string
18    val eventToString: event -> string
19  end
```

purposes. The interface is designed to be independent of the actual formalism at the most abstract level, which allows us to build tools that are formalism-independent. The entire interface can be seen in Listing 1.2. The interface defines the concepts of states and events (ll. 2–3). The most important functions are getInitialStates (l. 8) and nextStates (l. 11). getInitialStates returns the list of initial states. The reason that this is a list and not just a singleton state is to support non-deterministic formalisms. In addition to the state, we also return a list of enabled events for each initial state. The reason for this is that it makes it possible to optimize enabling calculation during depth-first traversal. nextStates takes as argument a state and an event and returns the successors using the same format as getInitialStates. If the given event is not enabled, the exception EventNotEnabled (l. 5) is raised. Additionally, the interface has a function for executing a sequence of events, executeSequence (l. 14), which works like nextStates, except it can execute zero, one, or more events rather than just one. Finally, the interface contains two functions, stateToString and eventToString (ll. 17–18) for converting states and events to a user-readable string.

State Representation. The interface in Listing 1.2 is formalism-independent. In order to instantiate the interface for CPN models, we need to define the types state and event, and define the functions in the interface.

As mentioned earlier, we need to be able to represent multiple states in a state space tool. To increase familiarity for previous users of the state space tool of CPN Tools [14], we define a structure Mark with data types and functions for manipulating states. We do not want to distinguish between the type used internally and the type manipulated by users in order to alleviate the need for translating between different representations, so the type should closely reflect the underlying CPN model. In Listing 1.3, we see (most of) the Mark structure for the model in Figs. 3 and 4. The type of the state is defined inductively in the hierarchy of the model. For each page, we define a record, which contains

Listing 1.3: New state representation.

```
1  structure Mark : sig
2    type Sender = {NextSend: NO ms}
3    type Network = {}
4    type Receiver = {NextRec: NO ms}
5    type Top = {A: NOxDATA ms, B1: NOxDATA ms, B2: NOxDATA ms, C1: NO ms,
6                C2: NO ms, D: NOxNO ms, Limit: UNIT ms, Received_1: DATA ms,
7                Received_2: DATA ms, Send: NOxDATA ms, Network: Network,
8                Receiver_1: Receiver, Receiver_2: Receiver, Sender: Sender}
9    type state = {Top: Top, time: time}
10   val get’Top’Receiver_1’NextRec : state -> NO ms
11   val set’Top’Receiver_1’NextRec : state -> NO ms -> state
12   val get’Top’Receiver_2’NextRec : state -> NO ms
13   val set’Top’Receiver_2’NextRec : state -> NO ms -> state
14   val get’Top’Receiver_1’B : state -> NOxDATA ms
15   val set’Top’Receiver_1’B : state -> NOxDATA ms -> state
16   ... (* several more accessor functions *)
17 end
```

entries for all places and sub-pages of the page. For example, in Listing 1.3 l. 2 we see the record defined for the Sender page in Fig. 4 (left). We see that we have only included “real” places, i.e., the four port places are not included so only the NextSend place is present. The type uses the names used in the model, and NextSend is thus represented using the record entry NextSend. The type of the NextSend is NO ms, i.e., multi-sets over the color NO of the place NO. The multi-set type is the same as used by CPN Tools. Similarly, types are defined for Network (l. 3), which contains no non-port places, and Receiver (l. 4), which contains one non-port place. The Top page is more complex (ll. 5–8), but uses the same structure. It contains entries for all non-port (i.e., all) places (ll. 5–6), but also entries for all sub-pages (ll. 6–8). The entries for sub-pages are named after the substitution transition and the type is that of the sub-page. For example, we see that the sub-page defined by the substitution transition Receiver 1 is represented by the entry Receiver_1 of type Receiver. Finally, at the top-level, we define the type of the state itself. As it is possible for a model to contain more than one top page, we define a new top level (l. 9), which contains all top pages (in this case just one entry Top of type Top). The state type also contains an entry for all reference declarations (in this model there are none) and the model time. As an example, we see the initial state of the network protocol in Listing 1.4.

State records, like the one in Listing 1.4, can be used as is, i.e., by using SML pattern matching or built-in accessor functions to pull values out of the record, or by building new structures with the correct names. For the user convenience,

Listing 1.4: Initial state of network protocol.

```
val initial = { Top = {
  A = empty, B1 = empty, B2 = empty, C1 = empty, C2 = empty, D = empty,
  Limit = 3‘(), Received_1 = 1‘"", Received_2 = 1‘"", Send = 1‘(1,"COLOUR")++
  1‘(2,"ED PET")++1‘(3,"RI NET"), Network = {}, Receiver_1 = {NextRec = 1‘1},
  Receiver_2 = {NextRec = 1‘1}, Sender = {NextSend = 1‘1} }, time = 0 }
```

we have also created set- and get-functions to access all pages and places of the structure. These functions all use the same naming convention, which is the function name (get or set) followed by a quote ('). Then comes the complete path to the place or page we wish to access, separated by quotes. The functions take a complete state as argument. Getter functions return either a multi-set of the appropriate type or a record describing the selected page. Setter functions instead take an additional parameter of the correct multi-set or record type and returns a new state, which is identical to the one given as the first parameter, except that the selected place/page marking has been replaced. Examples of setter and getter functions can be seen in Listing 1.3 in ll. 10–15. In addition to providing accessor functions for the "real" places represented in the state record, we also provide accessors which provide access to port and fusion places, so it is possible to use, e.g., get'Top'Receiver_1'B, to get the marking of the port place B in the receiver module. This function looks up the value on the corresponding socket place. This function is identical to get'Top'B1.

Event Representation. For events, we must make a choice between ease of use and compositionality. We first outline the obvious hierarchical approach to events and some of the problems of that. Then we describe our current implementation, which is not hierarchical (and thus does not as easily support compositionality).

The hierarchical event representation (Listing 1.5) is the natural companion to the state representation. Instead of types and records, we use structures and data types. For each page, we have a structure defining a data-type with a constructor for each transition and substitution transition. The type of each

Listing 1.5: Hierarchical representation of events.

```
structure Bind : sig
  structure Top : sig
    structure Sender : sig
      datatype event = Send_Packet of {n: INT, p: STRING}
                     | Receive_Acknow of {k: INT, n1: INT, n2: INT}
    end
    structure Network : sig
      datatype event =
        Transmit_Packet of
          {n: INT, p: STRING, success1: BOOL, success2: BOOL}
      | Transmit_Acknow1 of {n: INT, success1: BOOL}
      | Transmit_Acknow2 of {n: INT, success2: BOOL}
    end
    structure Receiver : sig
      datatype event =
        Receive_Packet of {k: INT, n: INT, p: STRING, str: STRING}
    end
    datatype event = Sender of Sender.event
                   | Network of Network.event
                   | Receiver_1 of Receiver.event
                   | Receiver_2 of Receiver.event
  end
  datatype event = Top of Top.event
end
```

constructor contains either a record with all variables (for normal transitions) or a reference to a previously defined data-type (for substitution transitions).

While this type definition is nice and natural, it has the major deficit that it is very cumbersome to use. The problem is that while data-type constructors are scoped, they are not context-sensitive. Thus, to refer to the transition Receive Acknow on the Sender page, we would need to write `Bind.Top.Sender Bind.Top.Sender.Receive_Acknow {k, n1, n2}`, and the verbosity and redundancy only gets worse if we have deeper hierarchies. We cannot solve this problem by opening all structures unless we require that all transitions, globally in the model, have unique names, and this is against the locality inherent in Petri nets.

Instead, we define a data-type as in Listing 1.6. We define a constructor for each transition named after the page it resides on and the name of the transition. The type of each constructor is a pair of an instance number and a record containing all variables associated with the transition. This definition is not as natural as the hierarchical one, and it re-introduces the "magic" instance numbers. To alleviate the introduction of instance numbers, we also define symbolic constants (ll. 10–14) for the path to each page instance. Using this, we can refer to the Receive_Acknow transition on Sender as `Bind.Sender'Receive_Acknow (Bind.Top.Sender, {k, n1, n2})`, where only Bind and Sender are repeated, and the latter only because the substitution transition has the same name as the page.

A final way to represent events is to create a data-type with a constructor for each transition instance, named after the path leading to the transition instance. While this is nice to use at first sight, it is even less compositional than both of the previous representations, and has the problem of making two instances of the same transition have completely different constructors.

3.3 Optimizations

A thing to notice about the representation of the state in Listing 1.3 is that it is immutable, i.e., that it is impossible to change markings of individual places

Listing 1.6: New representation of events.

```
1  structure Bind : sig
2    datatype event =
3      Network'Transmit_Acknow1 of int * {n: INT, success1: BOOL}
4    | Network'Transmit_Acknow2 of int * {n: INT, success2: BOOL}
5    | Network'Transmit_Packet of
6        int * {n: INT, p: STRING, success1: BOOL, success2: BOOL}
7    | Receiver'Receive_Packet of int * {k: INT, n: INT, p: STRING, str: STRING}
8    | Sender'Receive_Acknow of int * {k: INT, n1: INT, n2: INT}
9    | Sender'Send_Packet of int * {n: INT, p: STRING}
10   val Top : int
11   val Top'Network : int
12   val Top'Receiver_1 : int
13   val Top'Receiver_2 : int
14   val Top'Sender : int
15 end
```

in a state without creating a completely new state. This is a nice property we can use to make several optimisations. Immutability allows us to use the same representation internally as we expose to the user, as the user is not able to modify the representation. This has the great advantage that we do not need to translate between different representations in a state space tool (as happens in CPN Tools, where the exposed representation of a state is a Node, which is really an integer pointing into a mutable tree). Having the same representation internally and externally also lowers the barrier for users to become developers and experiment with more advanced aspects of state space reduction methods.

The implementation of the most interesting function from the interface in Listing 1.2, nextStates is implemented as in Listing 1.7. The setState function (not shown) basically copies the state record into the simulator. execute contains a large switch, which calls the correct CPN'occfun with the right parameters, and getState (not shown) reads the simulator representation and constructs a state record. The implementation is in fact slightly more intelligent. setState and getState keep track of the latest state record copied to/from the simulator. This improves performance a lot, in particular when doing depth-first traversal, as we will, most of the time, want to compute successors of a successor of the state currently stored in the simulator. As we have already calculated successors of this state and do not change it, the simulator is able to use locality to more efficiently calculate the desired successors. By exploiting immutability of the state record we can re-use parts of it to do even better by combining it with locality to implement BDD-like data-structure, which is essentially a faster but less memory efficient implementation of the tree-based storage of CPN Tools [2]. Assume we are given a state-record, e.g., the initial state from Listing 1.4. When we execute the Send Packet transition on Sender, we know (statically), that we can only change A and Limit on Top. We can thus re-use the representation of all other places at the top level and the representation of all sub-pages by making getState used in Listing 1.7 dependent on the event. This not only alleviates the need to transfer state from the simulator to the new state records, it also makes equality tests faster by reducing to pointer comparison for sub-pages and unchanged places. Furthermore, re-using old representations conserve memory. This does not ensure that we only store the multi-set 1‘1 once (and is hence not as memory efficient as the representation of CPN Tools), but on the other hand does not spend any time trying to unify multi-sets that are almost the same. This can also be exploited in the other direction. When asked to compute successors for a certain state, we only need to transfer pages and places that have actually changed (by changing the implementation of setState used in Listing 1.7). All of this can be done completely independently of the interface, without making explicit whether the interface is implemented in the most naive way or whether locality-optimisations take place (except for faster execution in the latter case).

Listing 1.7: Implementation of nextStates function.

```
fun nextStates (state, event) =
  (setState state; execute event; getState())
```

3.4 Auxiliary Functions

In order to provide the interface in Listing 1.2, we need to generate model-specific functions; basically the getState, execute, and setState functions used in Listing 1.7. Furthermore, we need to generate the Mark and Bind structures. The CPN simulator contains a set of tables, which can be used to inspect the model, but these tables are optimized for incremental syntax-check and fast simulation, and are therefore not very easy or fast to traverse. We have therefore developed an interface to the static part of the model, i.e., the pages with places, transitions, arcs, and all annotations of each. This interface can also be used for other purposes. We have already used it to generate model-specific hash-functions, marshaling of states and events, and ordering of states and events.

The generated hash-functions calculate hash values inductively in the structure of the model. We build "strings" on several levels, from multi-sets as strings of tokens (which may again be strings of simpler values), over pages as strings of places (multi-sets), to models as strings of pages. Using a simple combinator function which can calculate the hash value of a string given the hash values of each of it elements and hash-functions for all simple types, we can calculate a hash value for an arbitrary CPN model in a very efficient way. Furthermore, by using different combinator functions, we can efficiently generate multiple linearly independent hash functions. Such hash functions are useful for many things, such as putting states into hash tables (implementing full state space traversal), storing only a hash-value for each state (implementing hash compaction), or using the hash-value to set a bit in a bit-array (implementing bit-state hashing).

Marshaling is implemented using a strategy similar to the hash function. If we know how to store each character of a string, we can store the entire string by writing the length of the string and each character. Marshaling is useful for storing states to disk (implementing various disk-based state space traversal algorithms), or for transmitting states over a network (implementing distributed state space traversal).

Ordering is also implemented using the same strategy, by basically inductively defining a lexicographical order. Orders are useful for storing states to disk, as it is often useful to sort states when storing them on disk. It is also useful for storing states in search trees, which is used by many algorithms built into Standard ML, such as algorithms for calculating strongly connected components of graphs, which is useful for determining certain liveness properties of CPN models.

4 The Java CPN Model Interface

As mentioned in the introduction, many applications can benefit from tight integration with CPN models and the CPN simulator. If such applications are algorithmic in nature, we suggest using the SML interface described in the previous section, as it does not have the overhead of communication via TCP/IP. For most other applications, we propose that the Java interface described in this section is used as the overhead is irrelevant for many applications. The Java interface provides a high-level object-oriented representation of CPN models as well as an implementation of the protocol used by the CPN Tools graphical editor to communicate with the CPN simulator. As we furthermore provide an importer package that is able to read models created with CPN Tools, this interface makes it possible to create tools that load, manipulate, and simulate CPN models. Applications with these purposes often need to provide a user-friendly user interface or integrate with other applications. For these reasons, we have decided to create this interface in Java, which is widely used and provides many frameworks and tools for creating user-friendly applications.

4.1 Object Model

The CPN object model is a cleaned-up re-implementation of the model of the BRITNeY Suite [16], created for the ASAP model checking platform [11]. ASAP builds on the Eclipse platform [4], and so it is natural to use Eclipse frameworks for the implementation of the Java interface. In order to improve interoperability with other tools, we also support the ISO/IEC 15909-2 transfer format standardisation effort [7].

Our object model builds on version 1.1.5 of ISO/IEC 15909-2, in particular the *PNML Core Model* (Fig. 2 in [7]) and the *High-Level Core Structure* (Fig. 8 in [7]). In addition, we have added some extensions for CPN Tools specific features (to support CPN Tools' concept of time and code segments for transitions). In order to not pollute the basic model, we have basically implemented the PNML Core Model, and added features from the High-Level Core Structure and the CPN Tools specific extensions as add-ins. We have also extended the PNML Core Model with a simplified version of *Modular PNML* [9] to support hierarchical nets. The resulting object model can be seen in Fig. 5. Basically, we have a PetriNet at the top left corner. A Petri net can contain one or more Pages (middle left), which can contain any number of Arcs and Objects (middle). Objects are basically Places and Transitions (bottom). Additionally, objects can be Instances, which basically correspond to substitution transitions in CPN Tools. Objects can have any number of Labels (middle top), which are annotations, that correspond to initial markings, place types, arc inscriptions, names, guards (or conditions), code segments, and time inscriptions (middle from left to right). Places, transitions, and arcs each have one or more add-ins (classes with dark gray background), which basically allows them to have typed access to their annotations. Annotations also have an add-in, which makes it possible to store a structured version of the annotation as well as a plain text version. The Annotations package with the light gray background at the top right is basically an implementation of the High-Level Core Structure except that we have added Time and Code annotations. The white classes outside of this package basically implements the PNML Core Model. The Instance and ParameterAssignement are simplified versions of ModInstance and ParamAssign (renamed to remove abbreviations). The change is that where Modular PNML introduces a concept of modules and import nodes, we just use the already defined concepts of page and

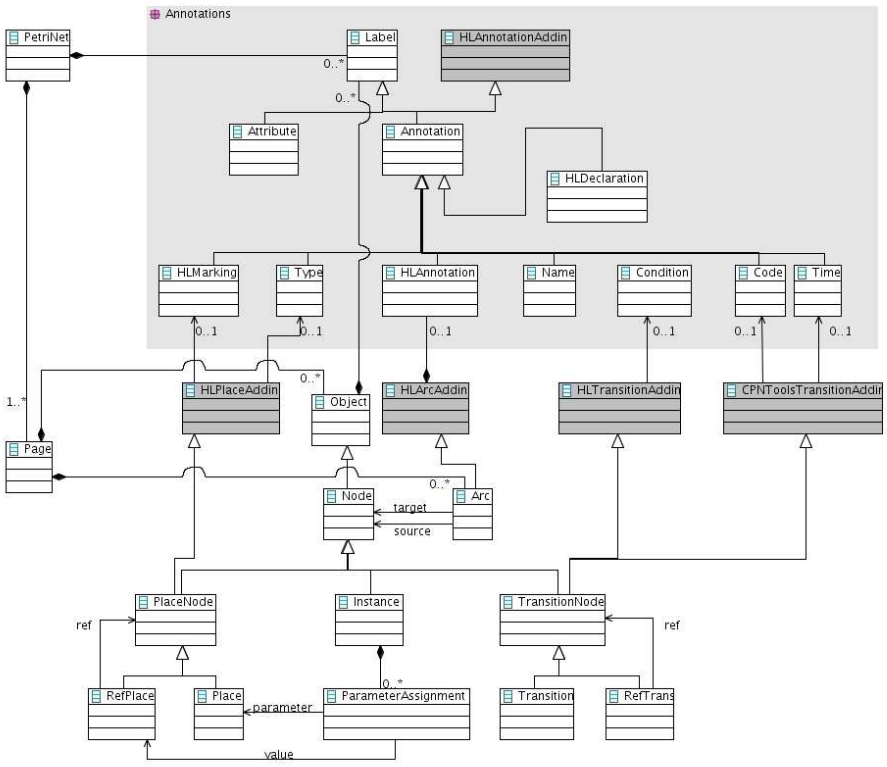

Fig. 5: Object model for CP-nets in the Java interface

place (as we only allow place-bordered modules). Furthermore, our Instance class is a Node and not just an object as CPN Tools allows arcs to and from substitution transitions. Finally, the place and transition add-ins do not contain their annotations (as they do in High-Level Core Structure), but just refer to them, as objects already contain labels and the add-ins merely provide typed access to these. We also have a few add-ins not shown in the figure. One adds an identifier to pages, arcs, labels and objects, and another adds names to pages and objects. Finally, we have an add-in for tool-specific information to Petri nets, objects, and labels.

The actual implementation of the object model is done using the Eclipse Modeling Framework (EMF) [5], which is a framework for implementing object models. EMF can generate implementation code from Java interfaces or from an UML diagram [13]. EMF is furthermore able to generate Java interfaces and UML diagrams from the model as well. In our case, we have described the model using Java interfaces, and the UML diagram in Fig. 5 is automatically generated from the model. In addition to automatic implementation, EMF also provides some nice features, such as automatic generation of XML marshaling and unmarshaling as well as an adapter functionality which is an extension of an observer architecture [6, Chap. 5]. This makes it possible to observe the object model for changes which is useful for editors, and to attach adapters adding new functionality to the classes.

CPN Tools Importer. Instances of the object model in Fig. 5 can be generated programmatically. It is of course desirable to create such models using a graphical user interface instead. For this reason we have created an importer, which allows programmers and users to import models created with CPN Tools.

The importer only imports the net-structure of the model but is prepared to support the graphical information as well, as we have made a preliminary implementation of the *Graphical Information* (Fig. 3 in [7]). All labels except for HLDeclarations are loaded as flat text; HLDeclarations use a structure similar to the *TermsUserDeclarations* (Fig. 17 in [7]), but the details are not shown here.

4.2 Protocol Implementation

The CPN Tools GUI communicates with the simulator process using a custom protocol. The protocol is an implementation of a remote procedure call (RPC) system [3, Chap. 5.3]. The protocol sends packets over a TCP/IP stream. Packets are transmitted in the custom BIS (boolean, integer, string) format, which is a binary packet format that basically takes care of marshaling of simple data types. Packets have an opcode which indicates the type of packet. CPN Tools primarily uses two opcodes, namely 1 (evaluate SML code) and 9 (RPC request). Packets with opcode 1 just contain a string to be sent for evaluation. Packets with opcode 9 have an additional integer to indicate which command to execute and sometimes another integer to determine a sub-command. Such commands must be combined in the correct way to syntax check an entire CPN model and generate simulator code for it.

In order to implement this protocol, one must implement the BIS packet format as well as high-level constructs translating to the lower-level command and sub-command integers, which is a tedious and error-prone job. Finally, we need to construct a component that can take a CPN object model and correctly send it to the simulator for syntax check and simulation. In Fig. 6 we see how this has been implemented in the Java interface. We see five packages. cpn.model represents the object model from Fig. 5, and cpn.model.importer is a package implementing an importer able to load a file created using CPN Tools. The class Job, which is outside of any of the packages, is part of Eclipse. The remaining three packages implement the protocol used to communicate with the CPN simulator. The classes are listed with the most high-level at the left. Only the classes at the top are meant to be used by most implementers. At the bottom-right, we have Packet, which implements the BIS package format. Such packets can be sent to a Simulator. The Simulator uses a delegate DaemonSimulator to communicate with the simulator via TCP/IP in the same way as CPN Tools. The Simulator class provides communication at the level of packets. The HighLevelSimulator provides stubs for all the calls supported by the simulator, and it is thus possible to communicate with named methods. It uses a PacketGenerator factory to actually create the packets it needs. The Checker class ties this to the object model hierarchy, and makes it possible to perform higher-level operations, such as syntax checking all declarations of a model. CheckerJob further lifts this and makes it possible to syntax check an entire net using a single call. The checker job integrates with the Eclipse platform and can provide feedback to the user. If this is not desired, one can use the simpler Checker class, which can be used independently of the platform used. For operations other than checking (such as simulation), one must go to the HighLevelSimulator. One will very rarely need to consider the Simulator, PacketGenerator, and their underlying classes.

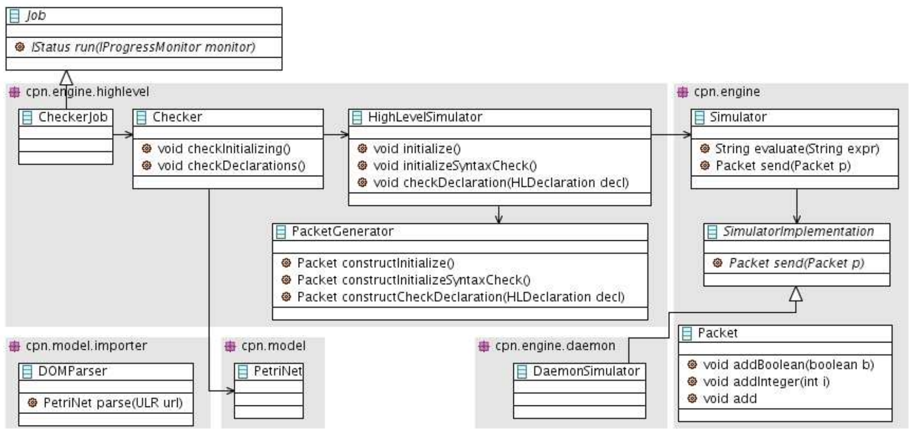

Fig. 6: Implementation of the protocol used to communicate with the simulator

5 Examples

In this section we show how to use the aforementioned interfaces by implementing a simple state space exploration tool that can check a model for dead-locks from the command-line. We first show the SML code implementing the traversal algorithm using the SML interface from Sect. 3, and then turn to the Java code for the command-line application loading a model and launching the exploration.

5.1 State-space Exploration

The implementation of the state space exploration algorithm can be seen in Listing 1.8. We actually implement an algorithm parametrised with a state property, so it is possible to check for other properties than dead-locks. The algorithm basically performs a recursive depth-first traversal of the state space and stores already expanded states in a hash-table. If a state not satisfying the property is found an exception is raised. The code starts (l. 1) by defining an exception to raise if a violating state is found. Then the built-in parametrised hash-function is instantiated. Then follows the implementation of the actual algorithm (ll. 6–35), which takes a predicate to apply to each state and a list of states from which to start the exploration. The function first defines the storage using SML's built-in HashTable (ll. 8). Then two mutually recursive functions dfs' and dfs'' are defined. dfs' (ll.20–31) traverses a list of states. It starts by checking if we have already traversed the state (l. 22), and, if so, continues with the next state (l. 23). If the state is new, it is stored (l. 25) and the predicate is checked (l. 26). If the predicated is violated, the exception is raised (l. 29). Otherwise we call dfs'' with the state before continuing with the rest of the states. dfs'' takes care of exploring successors resulting from executing all enabled events for a given state. It basically calculates successor states for each event (l. 14), and explores them using dfs' (l. 15) before traversing the rest of the events (l. 17). The entire function just calls dfs' with the given state(s). If no exception is raised, we return that no

Listing 1.8: Implementation of a simple state space exploration algorithm.

```
 1  exception Violating of CPNToolsModel.state

 3  fun combinator (h2, h1) = Word.<<(h1, 0w2) + h1 + h2 + 0w17
 4  val hash = CPNToolsHashFunction combinator

 6  fun dfs predicate states =
 7  let
 8    fun equals (a, b) = a = b
 9    val storage = HashTable.mkTable (hash, equals) (1000, LibBase.NotFound)

11    fun dfs’’ state [] = ()
12      | dfs’’ state (event::events) =
13        let
14          val successors = CPNToolsModel.nextStates (state, event)
15          val _ = dfs’ successors
16        in
17          dfs’’ state events
18        end

20    and dfs’ [] = ()
21      | dfs’ ((state, events)::rest) =
22        if Option.isSome (HashTable.find storage state)
23        then dfs’ rest
24        else let
25               val _ = HashTable.insert storage (state, ())
26               val violates = predicate (state, events)
27             in
28               if violates
29               then raise Violating state
30               else (dfs’’ state events; dfs’ rest)
31             end
32  in
33    (dfs’ states; (NONE, storage))
34    handle Violating state => (SOME state, storage)
35  end

37  fun none _ = false
38  fun dead (_, events) = List.null events
```

state violating the property was found, and the storage (l. 33). If an exception is raised, we also return the state violating the property. The last part of the listing contains a predicate that is never satisfied (l. 37) and one that checks for dead-locks (l. 38). The first is useful for performance testing, as it forces a full generation.

We have tested this implementation against the one built into CPN Tools. By varying the number of packets to transmit in the CPN model in Figs. 3 and 4 (altering the marking of the Send place) from two and upwards, we see that this implementation is 50-290 times faster (for 4-19 packets), discovers the same number of states as CPN Tools, and is able to explore larger state spaces than CPN Tools ($3.0 \cdot 10^6$ states when transmitting 25 packets compared to CPN Tools' $1.7 \cdot 10^6$ states when transmitting 19 packets).

5.2 Command-line State-space Analyser

To keep the example short, we use a simple implementation strategy. We load the model given as the first parameter, load the SML code shown in the previous example, which we assume is stored in a file simple-dfs.sml. Finally, we perform the exploration and show the result to the user. The implementation can be seen in Listing 1.9. We start by importing some classes needed (ll. 1–9). The rest of the code is the class implementing our state space tool. The class starts by obtaining the name of the file to analyse (l. 13). The file is loaded as a Petri net (l. 14), and we create a HighLevelSimulator. As we are running this outside of an Eclipse run-time environment, we need to supply a simulator manually. The simulator requires a delegate, which requires information about which host and port to connect to as well as the name of the run-time system to load. All of this takes place in ll. 16–18. If we are using the interface as part of an Eclipse application, we can just use the simplified version in l. 15, which obtains all parameters from a preference pane exposed to the user. We then create a new CheckerJob (l. 20), which requires a name (we just give it the name of the file), a Petri net, and a high-level simulator. We start (schedule) the job and wait for it to terminate (ll. 21–22). We then load the state-space algorithm developed previously (l. 23), and launch an exploration (ll. 24–30). We process the result of the exploration so the result we show the user is the violating state (if any) and the number of nodes explored. When we are done, we destroy the simulator (l. 32). This is needed as the simulator starts an external application, which should be shut down as well as a couple of Java threads for communication. By destroying the simulator we make sure to clean this up. If we quit the application (such as pressing the cross in a graphical application), this is performed automatically, but for this command-line application do this manually in order to terminate the program when the exploration is done.

The command-line tool can be executed as java StateSpaceTool protocol.cpn, and shows the first encountered dead-lock if there is one as well as the number of states stored.

Listing 1.9: Implementation of a command-line state space exploration tool.

```java
import java.io.File;
import java.net.InetAddress;
import java.net.URL;
import dk.au.daimi.ascoveco.cpn.engine.Simulator;
import dk.au.daimi.ascoveco.cpn.engine.daemon.DaemonSimulator;
import dk.au.daimi.ascoveco.cpn.engine.highlevel.HighLevelSimulator;
import dk.au.daimi.ascoveco.cpn.engine.highlevel.checker.CheckerJob;
import dk.au.daimi.ascoveco.cpn.model.PetriNet;
import dk.au.daimi.ascoveco.cpn.model.importer.DOMParser;

public class StateSpaceTool {
  public static void main(String[] args) throws Exception {
    String file = args[0];
    PetriNet petriNet = DOMParser.parse(new URL("file://" + file));
//  HighLevelSimulator s = HighLevelSimulator.getHighLevelSimulator();
    HighLevelSimulator s = HighLevelSimulator.getHighLevelSimulator(
        new Simulator(new DaemonSimulator(
          InetAddress.getLocalHost(), 23456, new File("cpn.ML"))));
    try {
      CheckerJob checkerJob = new CheckerJob(file, petriNet, s);
      checkerJob.schedule();
      checkerJob.join();
      s.evaluate("use \"simple-dfs.sml\"");
      System.out.println(s.evaluate(
        "let " +
        "  val (state, storage) = " +
        "    dfs dead (CPNToolsModel.getInitialStates()) " +
        "in " +
        "  (state, HashTable.numItems storage) " +
        "end"));
    } finally {
      s.destroy();
    }
  }
}
```

6 Conclusion and Future Work

In this paper we have described two interfaces to the CPN Tools simulator. One is very close to the simulator and written in Standard ML, and provides fast access to the simulator. The interface is useful for analysis methods and other algorithmic applications requiring little user-interaction. The other interface is written in Java and provides an object-oriented representation of CPN models, a means to import models created using CPN Tools, and high-level abstractions of the communication with the CPN Tools simulator, making it possible to integrate CPN simulation into Java applications, ranging from simple command-line applications to full-fledged graphical applications. Both of the interfaces are available to interested parties. Send an email to `ascoveco@cs.au.dk` for more information.

Future work includes replacing the current event implementation with the indicated hierarchical implementation from Listing 1.5. We can alleviate the syntactical problems by observing that while names of transitions may overlap, they rarely do in practise, so by just opening all structures, we can refer to the transition Receive Acknow. on the Sender page as `Top Sender Receive_Acknow {k, n1, n2}`. For transitions with overlapping names, we still need to use the very verbose naming, but we find that this is a reasonable price to pay for the more convenient representation.

The current Java interface only supports loading CPN models and syntax-checking them in one action. It would be useful to integrate the incremental syntax-checking capabilities of the simulator with the adapter functionality of the object model, so that whenever the object model is altered, it is automatically syntax-checked, independently of how the model is altered. This would be useful for editors, but also for applications generating models, as they are automatically checked for correctness and ready to be simulated.

References

1. ASCoVeCo Project webpage. Online: `www.daimi.au.dk/~ascoveco/`.
2. S. Christensen and L. M. Kristensen. State Space Analysis of Hierarchical Coloured Petri Nets. *Petri Net Approaches for Modelling and Validation, Lincom Studies in Computer Science 01*, pages 1–16, 2003.
3. G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems - Concepts and Design*. Addison-Weslay, 3rd edition, 2001.
4. Eclipse webpage. Online: `www.eclipse.org/`.
5. Eclipse Modelling Framework (EMF). `www.eclipse.org/modeling/emf/`.
6. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
7. ISO/JTC1/SC7/WG19. Software and System Engineering—High-level Petri nets—Part 2: Transfer Format, version 1.1.5.
8. J.B. Jørgensen and K.B. Lassen. Aligning Work Processes and the Adviser Portal Bank System. In *REBNITA'05*, 2005.
9. E. Kindler and M. Weber. A Universal Module Concept for Petri Nets—an implementation-oriented approach. *Informatik-Berichte*, (150), June 2001.
10. L.M. Kristensen, P. Mechlenborg, L. Zhang, B. Mitchell, and G.E. Gallasch. Model-based Development of a Course of Action Scheduling Tool. *STTT*, 10(1):5–14, 2007.
11. L.M. Kristensen and M. Westergaard. The ASCoVeCo State Space Analysis Platform: Next Generation Tool Support for State Space Analysis. In *Proc. of 8th CPN Workshop*, volume 584 of *DAIMI-PB*, pages 1–6, 2007.
12. L.M. Kristensen, M. Westergaard, and P.C. Nørgaard. Model-based Prototyping of an Interoperability Protocol for Mobile Ad-hoc Networks. In *Proc. of IFM'05*, volume 3771 of *LNCS*, pages 266–286. Springer-Verlag, 2005.
13. Object Management Group. Unified Modeling Language (UML), Version 2.1.1. Online: `www.omg.org/technology/documents/formal/uml.htm`, 2007.
14. A.V. Ratzer, L. Wells, H.M. Lassen, M. Laursen, J.F. Qvortrup, M.S. Stissing, M. Westergaard, S. Christensen, and K. Jensen. CPN Tools for Editing, Simulating, and Analysing Coloured Petri Nets. In *Proc. of ATPN'03*, volume 2679 of *LNCS*, pages 450–462. Springer-Verlag, 2003.
15. J.D. Ullman. *Elements of ML Programming*. Prentice-Hall, 1998.
16. M. Westergaard and K.B. Lassen. The BRITNeY Suite Animation Tool. In *Proc. of ATPN'06*, volume 4024 of *LNCS*, pages 431–440. Springer-Verlag, 2006.