The Access/CPN Framework: A Tool for Interacting with the CPN Tools Simulator[*]

Michael Westergaard[1] and Lars Michael Kristensen[2]

[1] Department of Computer Science, Aarhus University, Denmark
`mw@cs.au.dk`
[2] Department of Computer Engineering, Bergen University College, Norway
`lmkr@hib.no`

Abstract. Coloured Petri nets (CP-nets or CPNs) is a widely used formalism for describing concurrent systems. CPN Tools provides a mature environment for constructing, simulating, and performing analysis of CPN models. CPN Tools also has limitations if, for example, one wishes to extend the analysis capabilities or to integrate CPN models into external applications. In this paper we present Access/CPN, a framework that facilitates such extensions. Access/CPN consists of two interfaces: one written in Standard ML, which is very close to the simulator component of CPN Tools, and one written in Java, providing an object-oriented representation of CPN models, a means to load models created using CPN Tools, and an interface to the simulator. We illustrate Access/CPN by providing the complete implementation of a simple command-line state space exploration tool.

1 Introduction

Coloured Petri nets (CP-nets or CPNs) provide a useful formalism for describing concurrent systems, such as network protocols and workflow systems. CPN Tools [2] provides an environment for editing and simulating CPN models, and for verifying correctness using state space analysis. Sometimes, this is not enough, however. As CPN Tools is inherently graphical it cannot be controlled by external applications, so it is 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, describing a complex decision procedure as a CPN model for use in an application, and allowing users to set parameters of a model using a custom user interface. Due to the architecture of the simulator and state space tool in CPN Tools, it is also difficult to implement new analysis techniques like new more efficient state space methods.

CPN Tools basically consists of two components (see Fig. 1 (top)): a graphical editor (middle) and a simulator daemon (right). The editor allows users to interactively construct a CPN model that is transmitted to the simulator, which checks it for syntactical errors and generates model-specific code to simulate the

---

[*] Supported by the Danish Research Council for Technology and Production.

CPN model. The editor invokes the generated simulator code and presents results graphically. The editor can load and save models using an XML format (left in Fig. 1 (top)). The editor imposes most of the restrictions on the use of CPN Tools mentioned above. Replacing the editor with our own application, we can remove the limitations imposed by the editor. This has been done in several settings: In [10] simulation code is augmented with code to let it run within a web-server, allowing users to interact with the CPN model via a web-site to perform operational planning. In [8] the editor is replaced by a custom application to

Fig. 1. Architecture of CPN Tools (top) and Access/CPN (bottom)

allow military planning, and in [12] the editor is replaced by a more general-purpose application, which makes it possible to make domain specific visualisations of CPN models. Each of these examples use their own ad-hoc way to interact with the simulator. The simulator suffers from two problems making such interaction difficult. Firstly, the protocol used for communication between the editor and the simulator is 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.

In this paper we propose Access/CPN [1] which comprises two interfaces to the CPN simulator, alleviating the problems mentioned above. Access/CPN does not aim to replace CPN Tools as an editor for CPN models, but rather to allow researchers and developers to make experiments with the CPN formalism and use it as part of application development. Access/CPN has been developed as part of the ASAP model checking platform [9] and will be presented in this context, but is applicable also in general. One interface of Access/CPN is written in Java and the other in Standard ML (SML). Fig. 1 (bottom) shows how Access/CPN augments and replaces parts of CPN Tools. The Java interface (middle) consists of an object-oriented representation of CPN models, the ability to transmit this representation to the simulator and to perform simulation and inspection of the current state in the simulator. Furthermore, it includes a loader which can import models created using CPN Tools. The SML interface (right in Fig. 1 (bottom)) encapsulates the data-structures used in the simulator and provides an interface to a CPN model facilitating fast simulation, useful for efficient analysis and other applications executing transitions with little or no user-interaction.

In this paper, we first describe and exemplify the SML interface using a simple example, and then turn to the Java interfaces. The two parts can be read independently, and each assumes a knowledge of the language used. Finally, we conclude, compare with related work, and provide directions for further work.
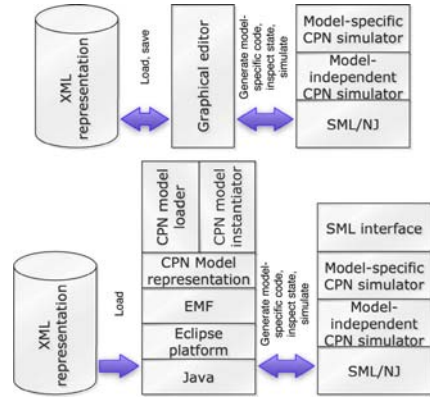
2   The SML CPN Model Interface

In this section we describe the SML interface of Access/CPN. The aim of the interface is to provide efficient access to the CPN simulator, in particular with the goal of implementing new and more 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 SML like the CPN simulator.

Example CPN Model. To exemplify the SML interface, we use a CPN model of a simple stop-and-wait protocol with one sender and two receivers as an example. The top module of the model can be seen in Fig. 2 (top), 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 has available capacity, the sender (Fig. 2 (bottom left)) transmits packets from Send onto the A place. The network (Fig. 2 (bottom middle)) transmits the packet to B1 and B2, optionally dropping one or both of the packets. The receivers (Fig. 2 (bottom right)) receive the packets on Received and transmit back acknowledgment onto C1 or C2, which the network transmits to D, optionally dropping one or both. When the sender receives acknowledgements from both receivers, the NextSend counter is updated and the transmission is repeated for the next packet. The model consists of four modules: Top, Sender, Network, and Receiver. The Receiver module is instantiated twice in the module Top corresponding to the substitution transitions Receiver 1 and Receiver 2.

2.1   Model Interface

The SML interface is designed with state space analysis in mind, but can be used for other purposes. It is designed to be independent of the actual formalism, making it possible to develop tools that are formalism-independent. The interface can be seen in Listing 1. It defines the concepts of states and events (ll. 2–3). The most important functions are getInitialStates (l. 7) and nextStates (l. 9). getInitialStates returns a list of initial states (in order to support non-deterministic formalisms, this is not restricted to being a single state) and a list of enabled events for each state. nextStates takes a state and an event, and returns successors using the same format as getInitialStates. If the given event is not enabled, the exception EventNotEnabled (l. 4) is raised. Additionally, the interface has a function for executing a sequence of events, executeSequence (l. 11), which works like nextStates except it executes any number of events, and two functions, stateToString and eventToString (ll. 13–14) for converting states and events to their string representation.

In addition to providing an implementation of the Model signature, the SML interface also provides utility functions like a hash-function, a partial order, and marshalling functions. Additionally, an interface for inspecting the model is provided, allowing users to create news model-specific functions, but not to modify the model; for this we suggest using the Java interface instead.
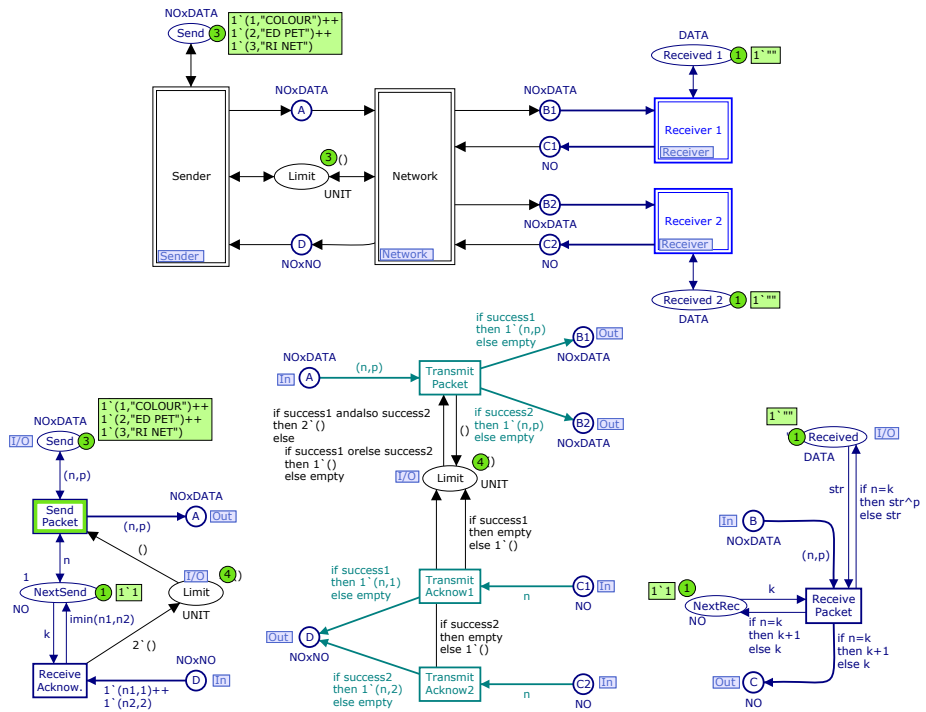
Fig. 2. Top (top), sender (bottom left), network (bottom middle), and receiver (bottom right) modules of a simple stop-and-wait protocol model with two receivers

Listing 1. Model interface

```
1  signature MODEL = sig
2      eqtype state
3      eqtype event
4      exception EventNotEnabled
5
6      val getInitialStates: unit -> (state * event list) list  (* Get initial states and enabled events *)
7      val nextStates: state * event -> (state * event list) list  (* Get successor states + enabled events *)
8      val executeSequence: state * event list -> (state * event list) list  (* Execute event sequence *)
9      val stateToString: state -> string  (* String representation of states *)
10     val eventToString: event -> string  (* String representation of events *)
11 end
```

Listing 2. 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, C2: NO ms, D: NOxNO ms,
6               Limit: UNIT ms, Received_1: DATA ms, Received_2: DATA ms, Send: NOxDATA ms, Network: Network,
7               Receiver_1: Receiver, Receiver_2: Receiver, Sender: Sender }
8    type state = { Top: Top, time: time }
9    val get'Top'Receiver_1'NextRec : state -> NO ms  val set'Top'Receiver_1'NextRec : state -> NO ms -> state
10   val get'Top'Receiver_2'NextRec : state -> NO ms  val set'Top'Receiver_2'NextRec : state -> NO ms -> state
11   val get'Top'Receiver_1'B : state -> NOxDATA ms   val set'Top'Receiver_1'B : state -> NOxDATA ms -> state
12   ... (* several more accessor functions *)
13 end
```

States. The interface in Listing 1 is formalism-independent. In order to instantiate the interface for CPN models, we need to define the types state and event. To increase familiarity for users of the state space tool of CPN Tools [2], we define a structure, Mark, with data types and functions for manipulating states. We want the type to reflect the hierarchical structure of the CPN model. Listing 2 shows (most of) the Mark structure for the model in Fig. 2. The type of the state is defined inductively in the hierarchy of the model. For each module, we define a record which contains entries for all places and sub-modules of the module. For example, in Listing 2 (l. 2) we see the record defined for the Sender module in Fig. 2 (bottom left). We see that we have only included real (non-port and non-fusion) places, i.e., so only the NextSend place is present. The type uses names from 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. The multi-set type is the same as used by CPN Tools. Similarly, types are defined for Network (l. 3) which contains no real places, and Receiver (l. 4) which contains one real place. The Top module is more complex (ll. 5–7), but uses the same structure. It contains entries for all real (i.e., all) places (ll. 5–6), and entries for all sub-modules (ll. 6–7). The entries for sub-modules are named after the substitution transition and the type is that of the sub-module. For example, we see that the sub-module defined by the substitution transition Receiver 1 is represented by the entry Receiver_1 of type Receiver. 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 module, we add a new top level (l. 8), containing all top modules (here Top), an entry for all reference declarations (here there are none), and the model time. For example, see the initial state of the network protocol in Listing 3.

State records, like the one in Listing 3, can be used with SML pattern matching, built-in accessor functions, and by building new structures. For convenience, we have also created set- and get-functions to access all modules and places of the structure. These functions use the naming convention: the function name (get or set) followed by the path of the place or module, separated by quotes. The functions take a state as argument, and getter functions return either a multi-set or a record describing the selected module. Setter functions also take a parameter of multi-set or record type and returns a new state like the one given with the selected place/module replaced. Examples of setter and getter functions can be seen in Listing 2 in ll. 9–11. In addition to providing accessors for the real places represented in the state record, we also provide accessors 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 (really B1 on the Top module).

Listing 3. Initial state of the network protocol example

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

Events. Events are defined by the data-type in Listing 4. We define a constructor for each transition named after the module it resides on and the name of

the transition (same naming convention as for accessor functions in the state representation). Each constructor is a pair of an instance number and a record with all variables of the transition. To alleviate the use of instance numbers, we define symbolic constants (l. 8) for the path to each module instance. Using this, we can refer to Receive_Acknow on Sender as `Bind.Sender'Receive_Acknow` (`Bind.Top.Sender, {k, n1, n2})`.

Listing 4. New representation of events

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

Listing 5. Implementation of a simple state space exploration algorithm

```
1  exception Violating of CPNToolsModel.state
2  fun combinator (h2, h1) = Word.<<(h1, 0w2) + h1 + h2 + 0w17
3  val hash = CPNToolsHashFunction combinator
4  fun none _ = false
5  fun dead (_, events) = List.null events

7  fun dfs predicate states =
8  let fun equals (a, b) = a = b
9      val storage = HashTable.mkTable (hash, equals) (1000, LibBase.NotFound)
10     fun dfs'' state [] = ()
11       | dfs'' state (event::events) = let val successors = CPNToolsModel.nextStates (state, event)
12                                           val _ = dfs' successors
13                                       in dfs'' state events
14                                       end
15     and dfs' [] = ()
16       | dfs' ((state, events)::rest) = if Option.isSome (HashTable.find storage state)
17                                        then dfs' rest
18                                        else let val _ = HashTable.insert storage (state, ())
19                                                 val violates = predicate (state, events)
20                                             in if violates
21                                                then raise Violating state
22                                                else (dfs'' state events; dfs' rest)
23                                             end
24  in (dfs' states; (NONE, storage)) handle Violating state => (SOME state, storage)
25  end
```

2.2   Example: State-Space Exploration

To illustrate the use of the SML interface, consider the implementation of a simple state space exploration algorithm in Listing 5 using the primitives presented above. The algorithm 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 first (l. 1) defines an exception to raise if a violating state is found, a built-in hash-function for states is instantiated (ll.2–3), and we define a predicate that is never satisfied (l. 4) and one that checks for dead-locks (l. 5). The rest is the actual algorithm, which takes a predicate to apply to each state and a list of states from which to start the exploration. The function defines the storage using SML's built-in HashTable (ll. 9). Then two mutually recursive functions dfs' and dfs'' are defined. dfs' (ll.15–23) traverses a list of states. It starts by checking if we have already traversed the state (l. 16), and, if so, continues with the next state (l. 17). If the state is new,

it is stored (l. 18) and the predicate is checked (l. 19), and an exception is raised (l. 21) on violation. Otherwise we call dfs″, which explores successors resulting from executing all enabled events for the given state using the nextStates primitive of the SML interface. dfs″ calculates successor states for each event (l. 11), explores them using dfs′ (l. 12), and traverses the remaining events (l. 13). The top function calls dfs′ with the given state (l. 24) and returns the storage. If a violation is found it is also reported.

3   The Java CPN Model Interface

Many applications can benefit from tight integration with the CPN simulator. For non-algorithmic applications, Access/CPN has a Java interface providing an object-oriented representation of CPN models, an importer to load models from CPN Tools, and an implementation of the protocol used to communicate with the CPN simulator. We have created this interface in Java as Java is widely used and provides many frameworks and tools for creating user-friendly applications.

Object Model. Our object model builds on version 1.1.5 of ISO/IEC 15909-2, in particular the *PNML Core Model* (Fig. 2 in [6]) and the *High-Level Core Structure* (Fig. 8 in [6]). We have extended the PNML Core Model with a simplified version of *Modular PNML* [7] to support hierarchical nets. The model is not shown here due to space limitations, but is a straightforward representation of a CPN model. Basically, we have a PetriNet containing one or more Pages, which can contain any number of Arcs, Places and Transitions, each containing appropriate Labels (e.g., name, place type, and arc inscription). The net structure is basically an implementation of the PNML Core Model, and Labels is an implementation of the High-Level Core Structure. Pages can also contain Instances, corresponding to substitution transitions in CPN Tools. Instances contain ParameterAssignments corresponding to port/socket assignments in CPN Tools, and are simplified versions of ModInstance and ParamAssign from Modular PNML. Implementation of the object model is done using the Eclipse Modeling Framework (EMF) [3], is a framework for implementing object models. EMF generates implementation code from Java interfaces and provides features like automatic generation of XML marshaling (saving/loading models as XML) and an adaptor architecture making it possible to observe the object model for changes and to add new functionality without changing the classes.

CPN Tools Importer.   Instances of the object model can be created programmatically, but it is desirable to create models using a graphical editor instead. For this reason we have created an importer, which allows programmers and users to import models created with CPN Tools. The importer imports the net-structure of the model and is able to load graphical information as well, as we have made a preliminary implementation of the *Graphical Information* from Fig. 3 in [6].

Protocol Implementation. The CPN Tools editor communicates with the simulator using a proprietary protocol, which is an implementation of a remote
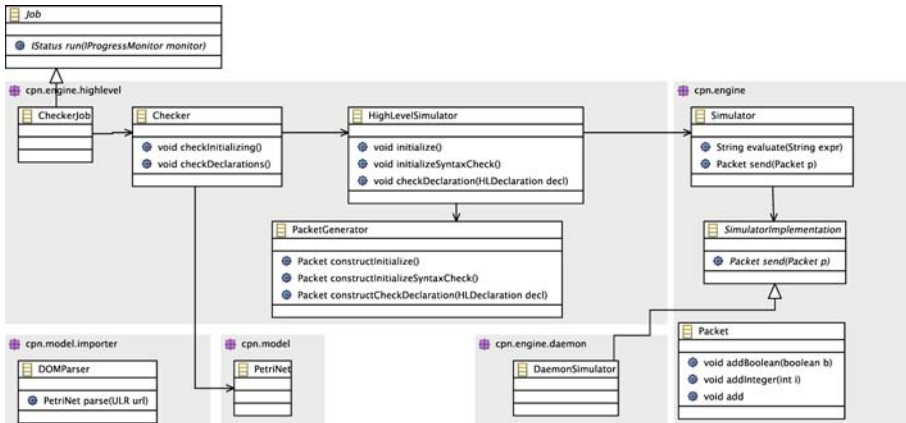
Fig. 3. Implementation of the protocol used to communicate with the simulator

procedure call (RPC) system. The protocol sends packets over a TCP/IP stream in the custom BIS (boolean, integer, string) format, which is a binary format that marshals simple data types. Packets have an integer op-code indicating the type of packet and some have an additional integer to indicate the command to execute. Commands must be combined in the correct way to syntax check a CPN model and generate simulator code for it. In order to implement this protocol, one must implement the BIS packet format, high-level constructs translating to the lower-level command integers, as well as a component that can take a CPN object model and send it all to the simulator for syntax check and simulation.

In Fig. 3 we see how the BIS protocol has been implemented in the Java interface of Access/CPN. It consists of five packages. cpn.model represents the object model described earlier, and cpn.model.importer contains the importer. 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 to the left. Only the classes at the top are meant to be used by application developers. 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 calls supported by the simulator, making it possible to communicate using named methods. It uses a PacketGenerator factory to create packets as needed. 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 CPN 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 undesired, one can use the simpler Checker class, which can be used independently of the platform used. For simulation, one

uses the HighLevelSimulator. One rarely needs to consider Simulator, PacketGenerator, and underlying classes.

3.1 Example: Command-Line State-Space Analyser

To illustrate the use of the Java interface, we implement a simple command-line application that uses the state-space algorithm from Sect. 2.2 to check models for dead-locks. We load a model given as a parameter, load the SML code shown previously, perform the exploration, and show the result to the user. The Java implementation can be seen in Listing 6. We start by importing classes needed (ll. 1–5). The code starts by obtaining the name of the file containing the CPN model to analyse (l. 9). The file is loaded as a Petri net (l. 10), and we create a HighLevelSimulator. As we are running this outside of an Eclipse application, we need to supply a simulator manually. The simulator requires a delegate, which must have information about which host and port to connect to as well as the name of the run-time system to load. All of this is handled in ll. 11–12. If we are using the interface as part of an Eclipse application, we can leave out the parameter in line 12. We then create a new CheckerJob (l. 13), which requires a name (here the string My model), a Petri net, and a high-level simulator. We start (schedule) the job and wait for it to terminate (l. 14). We then load the state-space algorithm developed in Sect. 2.2 (l. 15), and launch an exploration (ll. 16–18). We process the exploration result and show the user the violating state (if any) and the number of states explored. When done, the simulator is shut down (l. 20). The application can be executed as java StateSpaceTool protocol.cpn.

Listing 6. Implementation of a command-line state space exploration tool

```
1   import java.io.*; import java.net.*;
2   import dk.au.daimi.ascoveco.cpn.engine.Simulator; import dk.au.daimi.ascoveco.cpn.engine.daemon.DaemonSimulator;
3   import dk.au.daimi.ascoveco.cpn.engine.highlevel.HighLevelSimulator;
4   import dk.au.daimi.ascoveco.cpn.engine.highlevel.checker.CheckerJob;
5   import dk.au.daimi.ascoveco.cpn.model.PetriNet; import dk.au.daimi.ascoveco.cpn.model.importer.DOMParser;

7   public class StateSpaceTool {
8     public static void main(String[] args) throws Exception {
9       String file = args[0];
10      PetriNet petriNet = DOMParser.parse(new URL("file://" + file));
11      HighLevelSimulator s = HighLevelSimulator.getHighLevelSimulator(
12          new Simulator(new DaemonSimulator(InetAddress.getLocalHost(), 23456, new File("cpn.ML"))));
13      try { CheckerJob checkerJob = new CheckerJob("My model", petriNet, s);
14          checkerJob.schedule(); checkerJob.join();
15          s.evaluate("use \"simple-dfs.sml\"");
16          System.out.println(s.evaluate("let val (s, storage) = dfs dead (CPNToolsModel.getInitialStates())" +
17                                        " in (s, HashTable.numItems storage) " +
18                                        "end"));
19      } finally {
20          s.destroy();
21  } } }
```

4 Conclusion and Future Work

In this paper we have described Access/CPN, which provides two interfaces to the CPN Tools simulator. One is close to the simulator and written in Standard ML providing fast access to the simulator, which is useful for analysis methods and other algorithmic applications. 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 abstractions of the communication with the CPN Tools simulator, making it possible to integrate CPN simulation into Java applications. Access/CPN is currently used as part of the ASAP platform [9] and in a master's thesis on automatic code generation of CPN models [4]. Access/CPN has already been distributed to several interested parties and is available from [1].

The BRITNeY Suite [12], originally developed for visualisation purposes, resembles Access/CPN as it also allows programmers to interact with the simulator of CPN Tools. BRITNeY requires that programmers implement their programs as extensions of BRITNeY, whereas Access/CPN makes it possible to embed the CPN simulator into other programs, allowing greater freedom. In that respect, Access/CPN is a significant improvement over the interface provided by BRITNeY. The Petri Net Kernel (PNK) [11] shares many of the same traits as Access/CPN, i.e., a representation of Petri nets and ability to use Petri net simulators. PNK, like BRITNeY, however, makes programmers write their programs within PNK and is more focused on making it easy to make Petri net tools rather than using Petri nets within external applications.

As part of future work, it would be useful to integrate the incremental syntax-checking capabilities of the CPN Tools simulator with the object model, so when the object model is altered it is automatically syntax-checked and the simulation code is regenerated. This would be useful for editors and model generating applications. It would also be interesting to use Access/CPN for integrating CPNs into meta-modelling tools like PNK or the High-Level Architecture [5].

References

1. Access/CPN download, http://www.daimi.au.dk/~ascoveco/accesscpn/
2. CPN Tools webpage, www.daimi.au.dk/CPNTools/
3. Eclipse Modelling Framework (EMF), www.eclipse.org/modeling/emf/
4. Espersen, K.L., Kjeldsen, M.K.: Automatic Code Generation from Process-Partitioned Coloured Petri Net Models. Master's thesis, Dept. of Computer Science, University of Aarhus (2008)
5. Modeling and Simulation High Level Architecture. IEEE-1516
6. ISO/JTC1/SC7/WG19. Software and System Engineering—High-level Petri nets—Part 2: Transfer Format, version 1.1.5
7. Kindler, E., Weber, M.: A universal module Concept for Petri nets. In: Proc. des 8.Workshops Algorithmen und Werkzeuge fr Petrinetze, pp. 7–12 (2001)
8. Kristensen, L.M., Mechlenborg, P., Zhang, L., Mitchell, B., Gallasch, G.E.: Model-based Development of a Course of Action Scheduling Tool. STTT 10(1), 5–14 (2007)
9. Kristensen, L.M., Westergaard, M.: The ASCoVeCo State Space Analysis Platform. In: Proc. of 8th CPN Workshop. DAIMI-PB, vol. 584, pp. 1–6 (2007)
10. Lindstrøm, B.: Web-based interfaces for simulation of coloured petri net models. STTT 3(4), 405–416 (2001)
11. Weber, M., Kindler, E.: The Petri Net Kernel. In: Ehrig, H., Reisig, W., Rozenberg, G., Weber, H. (eds.) Petri Net Technology for Communication-Based Systems. LNCS, vol. 2472, pp. 109–123. Springer, Heidelberg (2003)
12. Westergaard, M., Lassen, K.B.: The BRITNeY Suite Animation Tool. In: Donatelli, S., Thiagarajan, P.S. (eds.) ICATPN 2006. LNCS, vol. 4024, pp. 431–440. Springer, Heidelberg (2006)