Access/CPN 2.0: A High-level Interface to Coloured Petri Net Models

Michael Westergaard[*]

Department of Mathematics and Computer Science,
Eindhoven University of Technology, The Netherlands
m.westergaard@tue.nl

Abstract. This paper introduces Access/CPN 2.0, which extends Access/CPN with high-level primitives for interacting with coloured Petri net (CPN) models in Java programs. The primitives allow Java programs to monitor and interact with places and transitions during execution, and embed entire programs as subpages of CPN models or embed CPN models as parts of programs. This facilitates building environments for systematic testing of program components using a CPN models. We illustrate the use of Access/CPN 2.0 in the context of business processes by embedding a workflow system into a CPN model.

1 Introduction

Coloured Petri nets (CPNs) have proved to be a useful formalism for different modelling tasks, including verification of network protocols and modelling of business processes. CPNs are a combination of Petri nets and a programming language, Standard ML. The use of a real programming language for inscriptions rather than a specially tailored inscription language has allowed usage of CPN models beyond modelling, effectively using the CPN modelling language as an implementation language. The aim of Access/CPN 2.0 is to allow programmers to use CPN models in programs in a more high-level manner, either by embedding a CPN model as a component of a program, by embedding a program as part of a CPN model, or by observing and interacting with places and transitions of CPN models during model execution. All a programmer has to do is adhere to an interface, and Access/CPN 2.0 takes care of all synchronization. Access/CPN 2.0 supports symmetric communication between CPN models and programs without polling on either side.

An early attempt of integrating coloured Petri nets is [5], where a library for communication between CPN models and (Java) components is devised with the purpose of invoking code written in other languages and vice versa by means of a generic remote procedure call mechanism. In [6] the authors use a CPN

model as the back-end of a web-service for simulating influence nets for operational planning using an ad-hoc means of communication, and in [4] a CPN model is used as server for a course-of-action tool for scheduling military operations using the general-purpose communication library COMMS/CPN [3] to implement a remote procedure call mechanism. Common for these solutions is that they make communication explicit in the model, potentially cluttering it. From another point of view, interaction has additionally been provided with visualisation libraries, such as MIMIC/CPN [10] and the successor, the BRITNeY Suite [15], which allow modellers to set up domain-specific visualisations of CPN models, letting users interact with models indirectly using domain-specific user-interfaces. Other tools, like Renew [11], allow Java inscriptions, making it directly possible to execute Java code when a transition is executed, at the cost of cluttering the model and allowing synchronous communication only. Most recently, the Access/CPN [13] library has been used to interact with CPN models using the simulator component of CPN Tools [2]. This allows scenarios where the simulation is directed by the user program or simulation results are shown to the user in a domain specific way without altering models. Unfortunately, the Access/CPN library is working at a fairly low level, such as executing a transition or getting the marking of a place, making the creation of such applications more demanding than necessary as the programmer has to handle synchronization manually. Access/CPN 2.0 aims to improve on these deficiencies by taking care of synchronization and not requiring cluttering models.

As a simple example of the use of Access/CPN 2.0 let us look at a CPN model of a simple workflow system (WS) in Fig. 1. At the top level (Fig. 1 (left)), we have two components, a Workflow System and a User, which communicate using
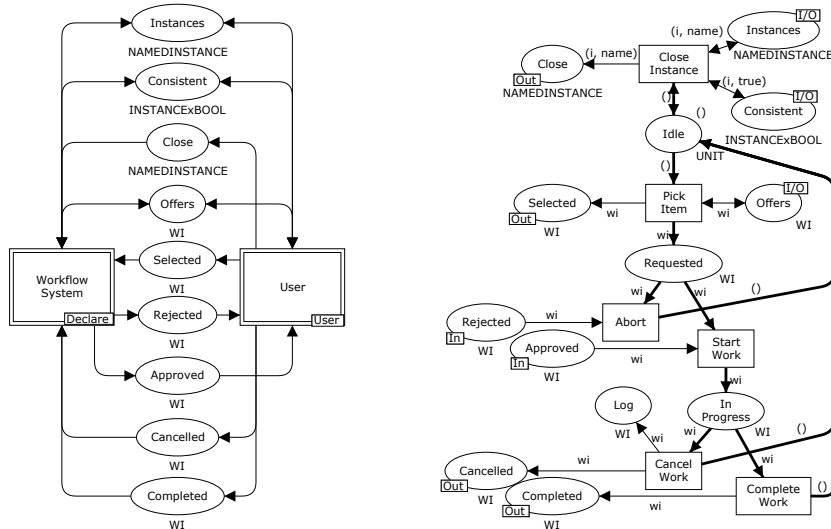

Fig. 1: CPN model of simple workflow system.

9 interface places. The interface is most easily described by its use, and in Fig. 1 (right) we see a simple CPN implementation of a User module. A user starts in the Idle state, and can choose to Pick Item. The work item (wi) is chosen from Offers, which is a read-only set of available items provided by the WS. After the user has picked an item, it is put on the Selected place to signal to the WS that it has been picked. The WS may either reject or accept a request for an item by moving the item to either Rejected or Accepted. If it is rejected, the user Aborts and returns to the Idle state, otherwise the user can Start Work and transition to the In Progress state. Here the user may either choose to Cancel Working on the item or to Finish Working, signalling the choice by moving the item to either Cancelled or Completed and returning to the Idle state. If work is cancelled, we log this so we can investigate why (by adding the item to the place Log). In the Idle state the user can additionally choose to close an instance of a workflow. Instances are provided by the read-only list Instances, and an instance can only be closed of it is Consistent (i.e., if all required items have been completed). Closing an instance is signalled by moving it to Close.

Our goal is to take the model in Fig. 1 and use it with a real workflow system; Access/CPN 2.0 reduces the effort needed to achieve such a task, and works on two levels: at the simple level, programs can interact with places and transitions during execution, and at the higher level, programs can act as submodules of CPN models or vice versa. Access/CPN 2.0 does this without requiring any annotation of CPN models, without having to focus on every detail of the simulation, and using real notification mechanism instead of polling.

Interaction between programs and places/transitions allows programs to be notified when a transition is executed or when the marking of a place is modified, similar to the monitoring facilities of CPN Tools. Access/CPN 2.0 implements these features in Java, which is more widely known than Standard ML. Access/CPN 2.0 additionally allows the programmer to modify the execution. In the example in Fig. 1, we could, e.g., monitor the Log place and store the logged data in a database for further processing, we could monitor the Start Work, Cancel Work, and Complete Work to measure efficiency of the user, or we could monitor and interact with the Pick Item transition to present a list of available work items to a user and even let the execution of the model depend on the user's choice.

Access/CPN 2.0 allows programs to embed CPN models or CPN models to embed programs. It can be useful to embed a CPN model as part of a program, e.g., if the program contains parts that are more easily described using a CPN model, or if no implementation is available yet and a stub implementation in the form of a model is used instead. It is useful to embed programs in CPN models if a CPN model is part of a larger system, where other components already exist either in other modelling languages or as real implementations. It is possible to directly use the existing component instead of recreating it as part of the model and potentially introducing errors and requiring extra effort. As a special case, it is possible to use CPN models for model-based testing of existing components: embed the component into a CPN model, model the environment

of the component as a CPN model, and use the model to exercise the component. In the example in Fig. 1, we would like to embed a real workflow system instead of having to model one. This could be useful for evaluating which of various strategies for picking work items results in the fastest execution.

The remainder of this paper is structured as follows: in the next section, we introduce the architecture of Access/CPN 2.0 and introduce the provided primitives. In Sect. 3, we show a couple of interesting applications of the primitives in the context of business process management and process mining. Finally, in Sect. 4, we conclude and provide directions for future work. Access/CPN 2.0 and video demonstrations are available from `cpntools.org/accesscpn/` and runs on all major platforms. All examples are included in ProM [9] nightly builds and installable in the upcoming ProM 6.1 release.

2   Architecture

In this section we introduce the architecture of Access/CPN 2.0 and the primitives offered. We regard Access/CPN 2.0 and Access/CPN as separate components the latter can be used without the Access/CPN 2.0 features. The architecture of Access/CPN 2.0 and required tools and
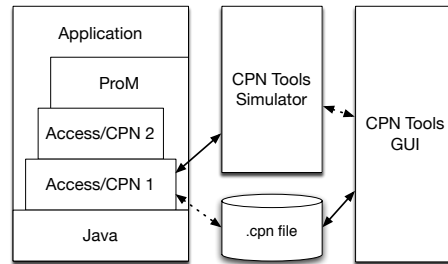

Fig. 2: Architecture of Access/CPN 2.0.

libraries is shown in Fig. 2. From right to left we have the CPN Tools GUI, the CPN Tools Simulator, and an application using Access/CPN 2.0. CPN Tools is used as the editor for CPN models and can read and write `.cpn` files. These files can be read by Access/CPN, represented in memory, and sent to the CPN Tools Simulator, which generates model-specific code to simulate the model. Access/CPN is able to communicate with the generated model-specific simulator code to execute transitions, and get and set the marking of places. Access/CPN 2.0 builds on top of Access/CPN. Instead of providing a CPN model specific interface like Access/CPN, Access/CPN 2.0 aims to provide a more abstract interface supporting monitoring places and transitions, and for embedding CPN models as parts of programs or vice versa. On top of Access/CPN 2.0 we have added integration with ProM [9], which additionally adds a GUI. The dotted line between the CPN Tools GUI and CPN Tools Simulator indicates that it is possible for the two components to communicate but not necessary for operation (models can just be generated or loaded from `.cpn` files without starting the CPN Tools GUI). Similarly, the line between the application using Access/CPN 2.0 and `.cpn` files is dotted to show we can but do not have to load a `.cpn` file. the CPN Tools GUI).

The main idea of Access/CPN 2.0 is to perform a *cosimulation* between a CPN model and Java classes, i.e., simulate the CPN model and run the Java classes at the same time, synchronizing when required. We have described a more generic cosimulation in [14], so the interface can be used by any modeling lan-

guage and programming language as long as they adhere to our interfaces. The cosimulation in [14] is focused on simulation, whereas we focus on using models together with real program. We we believe that CPN models and Java is the most natural combination for this, so we have focused on making this integration simple rather than implementing a generic interface. Naturally, interaction between the CPN simulator (written in Standard ML) and Java classes imposes an overhead, but we are not too concerned with performance, as we expect the usage to mostly concentrate on integrating one or more large components in a CPN model, so communication is limited.

To control the life cycle of Java classes participating in a cosimulation, we require that they implement the interface CPNToolsPlugin from Fig. 3 (ll. 1–3), containing methods that are called before and after a cosimulation. The start method takes a single parameter, an ExecutionContext, which is a key-value mapping used by plug-ins to store data between call-backs from the cosimulation and to communicate between plug-ins. The exact use of this is not dictated by Access/CPN 2.0. A main building block of the interface of Access/CPN 2.0 is a unidirectional channel (ll. 5–6). InputChannel allows clients to receive data by calling the getOffers method and OutputChannel allows sending data using offer.

```
01 interface CPNToolsPlugin {
02   void start(ExecutionContext context);
03   void end(); }

05 interface InputChannel { Collection<Value> getOffers(); }
06 interface OutputChannel { void offer(Collection<Value> offers); }

08 interface PlacePlugin extends CPNToolsPlugin, InputChannel, OutputChannel { }

10 interface TransitionPlugin extends CPNToolsPlugin {
11   boolean isEnabled(Binding binding, Number time);
12   Binding execute(Binding binding, Number time, Collection<String> rebindable); }

14 interface DataStore {
15   Collection<Value> getValues();
16   void setValues(Collection<Value> values); }

18 interface SubpagePlugin extends CPNToolsPlugin {
19   void setInterface(Collection<InputChannel> inputs, Collection<OutputChannel> outputs,
20                     Collection<DataStore> data);
21   boolean isDone(); }

23 interface CPNSimulation {
24   Collection<InputChannel> getInputs();
25   Collection<OutputChannel> getOutputs();
26   Collection<DataStore> getData();
27   void done(); }

29 interface CosimulationManager {
30   Cosimulation setUp(PetriNet model, Map<Instance<Page>, SubpagePlugin> subpagePlugins,
31                      Map<Instance<Place>, PlacePlugin) pPlugins,
32                      Map<Instance<Transition>, TransitionPlugin> tPlugins);
33   CPNSimulation launch(Cosimulation cosimulation);
34   void launchInCPNTools(Cosimulation cosimulation); }
```

Fig. 3: Access/CPN 2.0 API.

All values exchanged are of type Value, which comprises all values supported natively by CPN Tools. Access/CPN 2.0 defines three kinds of plug-ins and one interface for embedding CPN models: one plug-in for observing places (l. 8), one for observing transitions (ll. 10–12), and one to be plugged in instead of a subpage (ll. 18–21), as well as an interface representing the entire model (ll. 23–27).

A plug-in to observe places, a PlacePlugin (Fig. 3, l. 8), is a plug-in which implements the general super interface CPNToolsPlugin and the two channel interfaces. It acts as a bidirectional channel, and whenever a token is produced on the place associated with an instance of such a plug-in, it is offered to the plug-in (and removed from the model). Similarly, the plug-in can produce tokens on the associated place by offering them using getOffers. Tokens produced by a place plug-in are made available to the model as soon as the transition currently being executed (if any) is done, and tokens are removed from places and offered to place plug-ins immediately after execution of the transition producing them. Place plug-ins do not need to take explicit measures to achieve this synchronization.

A TransitionPlugin (Fig. 3, ll. 10–12), extends the common superinterface with two methods, isEnabled and execute. isEnabled is invoked before a transition is executed and can be used to reject executing a transition in a binding even though it is enabled in the underlying CPN model. execute is invoked when a transition observed by this plug-in is executed. Binding is part of Access/CPN and contains information about the transition instance executed and the binding of all variables surrounding it, providing all information needed to monitor an execution together with the time parameter, which describes when the transition was executed. To additionally allow modification of the execution, the parameter rebindable contains a collection of names of all variables that can be rebound by the plug-in. This means that by rebinding variables we can choose among different bindings of the transition. Formally, we strive for passing a set of rebindable variables such that, in the original model, if a binding b is enabled, then any binding b' with $b(v) = b'(v)$ for any v not in rebindable, is also enabled. We can approximate this statically by the set of free variables of a transition, i.e., the variables not occurring on any arc to the transition nor in the guard of the transition. Unfortunately, this rejects several variables we would like to rebind, namely any variable of a color set which is *large*, which roughly speaking means color sets containing more than 100 elements, thereby excluding integers and strings. This is a limitation of the simulator of CPN Tools, and as we would like to keep our models executable in CPN Tools without Access/CPN 2.0, we cannot work around that. Instead, we split a binding element up in a pre- and a postset, indicating tokens to remove when executing the binding and tokens to produce. We execute the preset of the binding element passed as parameter to execute and the postset of the binding element returned by the method. While altering the semantics of models, it works well in practise if we only rebind variables designed to be rebound, for example if variables are free except for being set to a value in the guard or if they are only bound using a double arc to a place. In our workflow example in Fig. 1, we would like to monitor and alter the execution of Pick Item. Here, we can safely rebind wi to any task available

on Offers even though wi is not a free variable. As with place plug-ins, transition plug-ins do not need to handle synchronization themselves; simulation is automatically paused while a transition plug-in is consulted.

A plug-in representing a subpage of a CPN model, a SubpagePlugin (Fig. 3, ll. 18–21), contains a method for defining the interface between the CPN model and the plug-in, setInterface, and a method allowing the plug-in to indicate when it is done processing, isDone. The interface consists of a collection of input parameters, a collection of output parameters, and a collection of data parameters. Input parameters correspond to input port places, i. e., the plug-in receives information from the surroundings by means of channels in this collection, so whenever a token is produced on the place in the CPN model corresponding to a parameter (a socket place), it can be received using the getOffers method of the input parameter. In the example in Fig. 1, the places Close, Selected, Cancelled, and Completed correspond to input parameters when using a subpage plugin for the Workflow System substitution transition. Output parameters play the same role for outputs and are in the example represented by the places Rejected and Approved. Data parameters are a restricted combination of input and output parameters. The plug-in is allowed to specify the values of the data store (and they correspond exactly to tokens on a corresponding socket place), but the CPN model is not allowed to modify that place. In our example, the places Instances, Consistent, and Offers correspond to data stores. Offering values on input parameters to subpage plug-ins, consumption of values produced on output parameters, and updates for datastores are guaranteed to be performed between transitions in the model. Subpage plug-ins may however need to guarantee that tokens are produced on places (corresponding to either output parameters or data stores) atomically, i.e., so no transition can be executed between the production on one place and on another. To achieve this, a subpage plug-in can synchronize on the ExecutionContext.

We do not provide an interface for plug-ins that can embed CPN models, but rather the other way around. The interface, CPNSimulation (Fig. 3, ll. 23–27) is the dual of the interface for adding subpages to a CPN model: it contains a way to get input and output parameters and data-stores of the model. The meaning of these is the same as for subpage plug-ins, except the channels now no longer correspond to socket places in the CPN model, but rather to port places on the top page. The dual of the isDone operation of the subpage plug-in is the done method indicating that the surrounding code is done with the CPN model, and Access/CPN 2.0 can stop simulating the model and free any allocated resources.

Access/CPN 2.0 provides a mechanism for setting up a cosimulation using plug-ins implementing the interfaces from Fig. 3 ll. 1–27 and executing it. This mechanism is shown in Fig. 3 ll. 29–34. setUp ties a CPN model together with place, transition and subpage plug-ins, and returns a Cosimulation, which can either be launched, resulting in a CPNSimulation which can be used by programs embedding CPN models, or the cosimulation can be launched in a special way that lets the user control the simulation directly from within the CPN Tools GUI (which is useful when embedding external code using subpage plug-ins).

3 Use Cases

Whereas the interfaces offered by Access/CPN 2.0 may seem simple, they are powerful in practise. We have implemented several CPNToolsPlugins and a ProM [9] plug-in allowing us to interactively construct a cosimualtion for execution. These examples, aside from the last one, are not intended to provide completely new functionality (for example, monitoring has previously been described in [7] and ProM orchestration in [1]), but just to give an idea of what is possible with Access/CPN 2.0, and the complexity of performing such tasks.

Showing Simulation Feedback. Sometimes we just want to run a simulation and see the end result, e. g., if the model implements a computation. For displaying results, we create a place plug-in, alerting users when a token is produced on observed places, and a transition plug-in reporting executed transitions.

Generating Execution Logs. Process mining is concerned with generating models from execution logs. One way of validating the usefulness of a process mining algorithm is to create a model, run executions of the model, and do process mining on the resulting data, comparing the result with the original model. We can easily do this using Access/CPN 2.0 by creating a transition plug-in which creates a log in the ExecutionContext and adds transition executions whenever they occur.

ProM Orchestration. ProM consists of many plug-ins, each of which consumes input values and produces output values. Normally, a user manually executes plug-ins but for complicated or repetitive tasks it is desirable to be able to automate this. A way of doing so is to use a CPN model do describe the workflow inside ProM and make sure that ProM executes plug-ins corresponding to elements of the model. We can do so using a transition plug-in and Access/CPN 2.0, identifying transitions of the CPN model with plug-ins of ProM and tokens with values. We can use mapping in the ExecutionContext and the binding of the transition to compute the actual parameters to pass to a plug-in when executing a transition, and rebind variables on out-going arcs according to the token/value mapping.

Embedding the Declare Workflow Engine and Operational Support. It would be nice to be able to use a real workflow engine directly from within a CPN model like the one in Fig. 1. We have a workflow engine called Declare, which allows us to specify workflows declaratively, and we would like to embed that in our model at it allows to focus on the task at hand, such as evaluating different strategies for picking tasks rather than modeling a workflow system. ProM supports operational support, i. e., on-line process mining aiming to aid a user with exactly that kind of decision. Operational support algorithms can be implemented in many ways. By embedding a model of an operational support algorithm, we can make it available to clients as if it was natively implemented. We can combine a plug-in embedding Declare with a plug-in embedding ProM's operational support engine into a single suite for evaluating operational support algorithms.

3.1 Summary

Table 1 summarise the plug-ins mentioned in this paper along with a classification of the type of plug-in and the complexity of the plug-in measured by lines of code (LoC) excluding boiler-plate code to illustrate that with Access/CPN 2.0 and relatively little effort, nontrivial tasks can be solved. We have provided only the LoC as a metric for effort put into the plug-ins, as most of the described plug-ins have been developed together with Access/CPN 2.0, and therefore the time spent on each individual plug-in is nontrivial to measure. One plug-in, ProMOrchestrate, was developed on its own, and took about 4 hours in one sitting to develop from scratch to completion.

Table 1: Overview of plug-ins.

Name	Type	LoC
InputValue	Place	1
ShowValue	Place	1
ShowTransition	Transition	38
GenerateLog	Transition	88
ProMOrchestrate	Transition	254
Declare	Subpage	403
OSServer	Superpage	200

4 Conclusion and Future Work

We have presented Access/CPN 2.0 and applications of it, primarily within the domain of business process management, but from the diversity of the examples we believe they support the general applicability of Access/CPN 2.0.

Many previously available libraries for communication between CPN models and external code work on a relatively low level, either providing communication primitives [3, 4, 6], remote procedure call inspired interfaces of more or less generality [5, 10, 15] or detailed access to the simulator [13]. In [12] we outlined a method for declaratively tying a model to an external program by separating the control between internally executed transitions (the model) and externally (the program), but the synchronization is either complicated and specified imperatively, or limited and domain-specific. In [14] we extended this idea to allow components to be cosimulated. In this case we used CPN modules and SystemC models as components. The high-level architecture [8] has similar aims for general simulation engines. Both of these interfaces are more focused on simulation, whereas we are more interested in using CPN models as parts of actual programs.

Using a CPN model as part of an implementation is a nice way to go quickly from a specification to a prototypical implementation, but for some applications the overhead of simulating a CPN model including the communication overhead may be too large. Therefore, approaches exist to automatically or semi-automatically generate template code for an implementation of a modelled system. It would be nice to fuse the functionality of Access/CPN 2.0 with such a code generator, allowing the generated code to directly interact with the plug-ins used by the model, thereby (mostly) filling in the templates generated from

the model. Future work also includes evaluating the overhead imposed by the cosimulation, e.g., in comparison with the monitoring facilities in CPN Tools.

References

1. L. Cabac and N. Denz. Net Components for the Integration of Process Mining into Agent-Oriented Software Engineering. In *ToPNoC*, volume 5100 of *LNCS*, pages 86–103. Springer, 2008.
2. CPN Tools webpage. Online: `cpntools.org`.
3. G. Gallasch and L.M. Kristensen. A Communication Infrastructure for External Communication with Design/CPN. In *Proc. of Third CPN Workshop*, volume 554 of *DAIMI-PB*, pages 79–93, 2001.
4. 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.
5. O. Kummer, D. Moldt, and F. Wienberg. Symmetric Communication between Coloured Petri Net Simulations and Java-Processes. In *Proc. ATPN'99*, volume 1639 of *LNCS*, pages 690–690. Springer, 1999.
6. B. Lindstrøm and L.W. Wagenhals. Operational Planning using Web-Based Interfaces to a Coloured Petri Net Simulator of Influence Nets. In *Proc. of Formal Methods Applied to Defence Systems*, volume 12 of *CRPIT*, pages 115–124, 2002.
7. B. Lindstrøm and L. Wells. Towards a Monitoring Framework for Discrete-Event System Simulations. In *Proc. of WODES'02*, pages 127–134. IEEE Comp. Soc. Press, 2002.
8. K.L. Morse, M. Lightner, R. Little, B. Lutz, and R. Scrudder. Enabling Simulation Interoperability. *Computer*, 39(1):115–117, 2006.
9. Process mining webpage. Online: `processmining.org`.
10. J.L. Rasmussen and M. Singh. *Mimic/CPN. A Graphical Simulation Utility for Design/CPN. User's Manual*. Online: `www.daimi.au.dk/designCPN`.
11. Renew webpage. Online: `renew.de`.
12. M. Westergaard. Game Coloured Petri Nets. In *Proc. of 7th CPN Workshop*, volume 579 of *DAIMI-PB*, pages 281–300, 2006.
13. M. Westergaard and L.M. Kristensen. The Access/CPN Framework: A Tool for Interacting With the CPN Tools Simulator. In *Proc. of ATPN'09*, volume 5606 of *LNCS*, pages 313–322. Springer, 2009.
14. M. Westergaard, L.M. Kristensen, and M. Kuusela. A Prototype for Cosimulating SystemC and Coloured Petri Net Models. In *Proc. of 10th CPN Workshop*, volume 590 of *DAIMI-PB*, pages 1–19, 2009.
15. M. Westergaard and K.B. Lassen. The BRITNeY Suite Animation Tool. In *Proc. of ATPN'06*, volume 4024 of *LNCS*, pages 431–440. Springer, 2006.