CPN Tools 4: Multi-formalism and Extensibility

Michael Westergaard[1,2,⋆]

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

[2] National Research University Higher School of Economics,
Moscow, 101000, Russia

Abstract. CPN Tools is an advanced tool for editing, simulating, and analyzing colored Petri nets. This paper discusses the fourth major release of the tool, which makes it simple to use the tool for ordinary Petri nets, including adding inhibitor and reset arcs, and PNML export. This version also supports declarative modeling using constraints, and adds an extension framework making it easy for third parties to extend CPN Tools using Java.

1 Introduction

CPN Tools [2] is a popular tool for modeling and analysis of colored Petri nets [6] (CP-nets). The large user base and several years of development has resulted in a stable and versatile tool for users and researchers working with CP-nets. CPN Tools incremental syntax check of models, making it very accessible for beginners, and provides tools for analysis both by means of simulation and state space generation, making it useful for research and industry alike. Unfortunately, some design choices made when starting CPN Tools development make it difficult to use for developers and researchers working with other formalisms similar to CP-nets. Furthermore, the modeling power of CP-nets imposes a certain mental overhead which is not desirable when performing simple modeling tasks. In this paper, we present the 4th major version of CPN Tools, which aims to make CPN Tools even more useful for regular users, and also provide developers and researchers with a solid base for simple extension development.

Colored Petri nets extend basic Petri nets (also known as place-transition Petri nets or PT-nets) [3] with distinguishable tokens and data types. This makes it possible to share net structure by relying on inscriptions. CP-nets constitute a pure extension and can hence simulate PT-net models. This often incurs a penalty in modeling complexity, however. CPN Tools 4 alleviates this by assuming sensible defaults for all inscriptions, which makes it possible to make a PT-net model in CPN Tools with the same syntax and amount of work required in a native PT-net editor.

⋆ Support from the Basic Research Program of the National Research University Higher School of Economics is gratefully acknowledged.

CPN Tools 4 also allows users to use the full syntax of the Declare [17] workflow language. This language does not explicitly focus on the control- or data-flow of models, but instead allows users to specify requirements on the order of execution of actions (transitions) such as "transition A cannot be executed before transition B" or "it is not possible to execute both transitions A and B". This is very useful for abstract specifications, especially in early phases, where the exact control-flow is of less significance. Some Declare constraints are very verbose to express explicitly, including the constraint stating that "it is not allowed to execute any transitions before A". Thus, by embedding the Declare language, CPN Tools 4 makes it possible to use the tool earlier in the development process, and later specify the declarative requirements more explicitly using classical net constructions or just retain the declarative specifications, as CPN Tools allows freely mixing of CP-nets, PT-nets, and Declare constraints.

Many low-level variants of Petri nets include extended syntax to extend the expressivity of the formalisms. CP-nets do not need these extensions as they can be expressed using common and documented patterns [1], but often these extensions constitute a shorthand which may be useful for recognition and ease of modeling. By embracing low-level nets, we think it is beneficial to support some of these extensions in CPN Tools as well. The focus has been on adding the most useful extensions in the most conservative way. We aim to make sure that a model working in a previous version of CPN Tools also works in future versions, and hence we prefer not to add extensions we are not sure will stand the test of time. For this reason, CPN Tools 4 adds support for inhibitor arcs and reset arcs in a limited form. We only allow an all-or-nothing semantics, which is contrary to the colored nature of CP-nets. The reason is that the semantics of colored inhibitor or reset arcs is not completely obvious, and we would prefer modelers to get experience with the limited versions before we extend support.

CPN Tools comprises two components, the user interface and the simulator, communicating using TCP. The simulator is developed is the functional language Standard ML, and the GUI is developed in the object oriented language BETA. While both languages are perfectly suitable for their uses in CPN Tools, they are not widely known. We have received a lot of requests to allow third party developers to contribute to CPN Tools. This includes researchers wanting to add their experimental extensions to CPN Tools instead of creating a tool from scratch, developers creating amazing extensions making the life easier for themselves and others, and our own students making projects of various complexity within a popular framework. While both components of CPN Tools are open source and the protocol between them public and documented, this is not an easy task due to the language barrier. CPN Tools 4 aims to make this easier by providing a back-end hook into the tool for Java developers.

CPN Tools has a history for allowing extensions. It uses a full programming language for inscriptions, which allows developers to develop libraries extending the annotation language of CPN Tools, including providing communication primitives using COMMS/CPN [4]. DESIGN/CPN, a predecessor of CPN Tools, even allowed such libraries to create elements elements in the GUI, leading to libraries such as a message-sequence chart library and Mimic/CPN [10], which is a general-purpose tool for creating and animating graphical elements from CP-net models. CPN Tools has had external extensions, including the BRITNeY Suite [16], providing model visualizations in an external tool, and the Access/CPN library [13, 15] making it possible to interact with the simulator from external Java programs. While these tools have made it easier to *interact* with CPN Tools, they have not made it possible to *extend* CPN Tools aside from useful but simplistic annotation extensions. Simulator extensions in CPN Tools provide an architecture for directly extending CPN Tools without having to bother with the relatively unknown languages BETA and Standard ML.

An obvious way to allow extensions of CPN Tools is to provide a high-level macro language. This would be similar to providing a domain-specific inscription language, however. While suitable for some cases, it would not allow the truly creative uses of CPN Tools in the past. Instead, we have chosen to provide powerful low-level primitives in a regular and well-known language, The idea is to add hooks into the simulator allowing developers to modify the foundational behavior as necessary. Furthermore, we have added hooks making it possible to draw and control graphical elements directly in the CPN Tools GUI. Simulator extensions can also directly interfere with the syntax check and simulation of models, allowing them to extend the semantics of CP-nets. Extensions can seamlessly support new operations to the protocol between the CPN Tools GUI and simulator, making it easier to add new functionality to CPN Tools, as functionality can be prototyped in Java and subsequently be moved to Standard ML for improved performance if required. Finally, extensions can interact directly with the model, making it possible to create completely new CP-net-like formalisms inside CPN Tools without having to resort to esoteric languages.

Simulator extensions can work on many levels. They can provide extensions to the CPN inscription language written in Java, similar to what was previously possible using Standard ML libraries. Extensions can create and manipulate graphical elements in the CPN Tools GUI as was possible in Design/CPN. Extensions can also filter the communication between the GUI and simulator, similarly to what the BRITNeY Suite previously offered [12], but much more tightly integrated with CPN Tools and easier to use. Extensions are also able to provide completely new functionality to the GUI, making it easier to implement certain features, including the Declare support, web-services integration, and PNML export (as specified in ISO/IEC 15909-1) of CPN Tools 4. Simulator extensions are designed to complement, not to replace, the Access/CPN library. While extensions can present themselves as separate applications, the intention is that they present themselves inside the CPN Tools GUI and not as separate applications.

In the remainder of this paper, we go though the major new features of CPN Tools 4. We first summarize the existing and new architecture of CPN Tools in Sect. 2, and turn to the multi-formalism extensions of CPN Tools in Sect. 3. In Sect. 4, we present the simulator extensions framework and present several examples of its use. Finally, we conclude and provide directions for future work.

2 Architecture

CPN Tools and Design/CPN before it have a bi-process architecture. This means they have a user-facing graphical user interface (GUI) and a lower level simulator component. The simulator component is responsible for the heavy algorithmic lifting whereas the GUI is responsible for allowing the user to indicate what the model should look like. While CPN Tools tries to hide this fact from the user, it can be useful to know. For example, this means that it is possible to design model on a relatively low-powered workstation and do heavy analysis on a more powerful grid, cloud or distributed architecture. This architecture is also important to understand simulator extensions in CPN Tools 4.

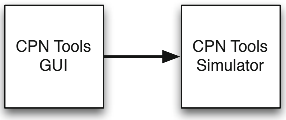


Fig. 1. Basic architecture of CPN Tools

The basic architecture of CPN Tools is shown in Fig. 1. Here, we see that the GUI is communicating directly with the simulator to provide editing and simulation of CP-net models. Here, the GUI initiates communication. Several tools have exploited this architecture to extend CPN Tools. Probably most prominent is the BRITNeY suite, which provided two modes of extensions; either it could be called as an external application to provide visualization, Fig. 2 (top), or it could mediate the communication between the GUI and the simulator (Fig. 2 (bottom)), allowing it to provide an even tighter integration such as performing inspection and on-the-fly modification of the constructed model and injection of appropriate inscription extensions.

Several extensions to CPN Tools have been proposed, both before and after the BRITNeY Suite, typically using the architecture in Fig. 2 (top) [4,8,9]. Access/CPN instead replaces the user interface component in Fig. 1, and this architecture has also been used in ASAP [14]. Mimic/CPN made it possible using Design/CPN to provide an architecture similar to Fig. 2 (top) with bidirectional communication between the GUI and simulator.

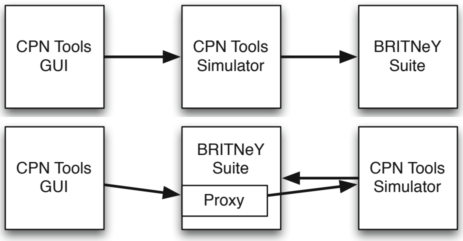


Fig. 2. CPN Tools and the BRITNeY Suite running in slave mode (top) and in filter mode (bottom)

With simulator extensions, we considered the least intrusive architecture change making it possible to reuse as much as possible of the existing code base, both ours and that developed by others. We wanted to make the extensions efficient and as far as possible transparent to end-users. We considered adding another process communicating with the user interface. The argument for this architecture is that it does not require (substantial) changes to the simulator component and it does not impose any overhead if unused. In the end, we decided to go with the architecture in Fig. 3. The main reason is that we do not wish to duplicate any implementation in Access/CPN and we want to be able to load and simulate any model created in CPN Tools using Access/CPN. We see that we add a new process for handling the extensions, much like the architecture in Fig. 2 (top). We allow bi-directional communication between the simulator and the GUI, as well as between the GUI and the extension manager. We also allow communication between the GUI and extension manager, but this is always mediated by the simulator. This architecture minimizes changes in the CPN Tools GUI, and the main challenge is a purely technical one, namely that the CPN simulator is inherently single-threaded. The architecture imposes minimal overhead as communication with the extension server only happens when communication with the simulator happens anyway.

CPN Tools uses an extensible protocol framework for communication between the GUI and the simulator, and we have extended this to also handle extensions and use the same framework for communication between the simulator and the extension manager. We extend the framework to handle call-back messages in the GUI (allowing the simulator or extensions to invoke procedures in the GUI) and to allow extensions to filter the communication between the user interface and the simulator.

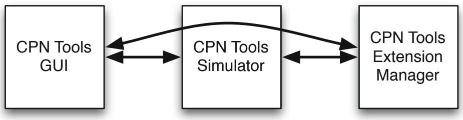


Fig. 3. Architecture of CPN Tools 4

3 Multi-formalism Support

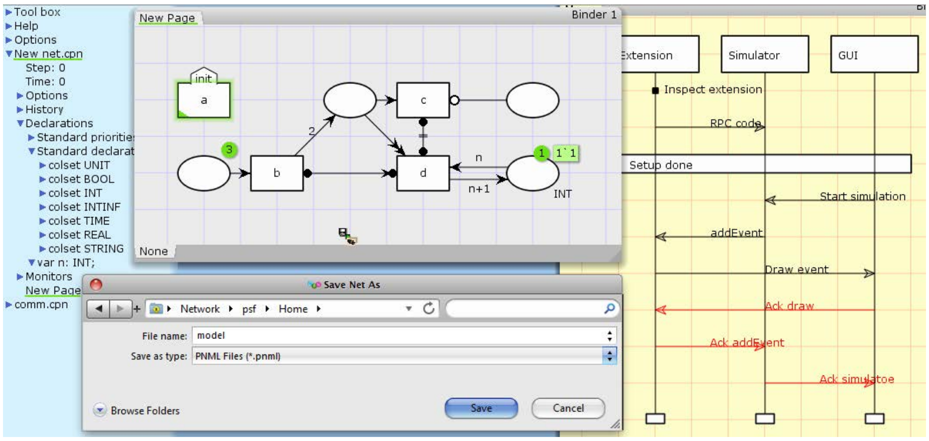
CPN Tools 4 extends CP-nets with provisions for directly handling PT-nets [3] and Declare [17] models in addition to CP-net models. We do this in a conservative way, meaning that each formalism can be used completely independently of any of the others and no formalism imposes an overhead when not used. PT-nets can easily be embedded in CPN models and are handled by introducing syntactical sugar. We also introduce modeling extensions that traditionally extend the power of low-level formalisms but are just conveniences for CP-nets. Declare models deal only with the ordering of tasks, and have minimal or no handling of data flow. As such, we can see the join of a CP-net model and a Declare graph model as the synchronous product of the behavior of the CP-net model (projected onto just the transition instances) and the Declare model. Thus, Declare constraints are purely restrictions of the dynamic behavior of CP-nets, similar to the concept of time used in CPN Tools.

3.1 PT-Net Support

High-level net modeling formalisms, such as CP-nets, easily embed lower-level net formalisms, such as PT-nets. This is traditionally done by introducing a color set, or type, with just one element. In CPN Tools this type is called UNIT and it contains a single value, (). By making CPN Tools automatically recognize no explicit type as UNIT and no arc inscription as (), it is easy to emulate most common PT-net behavior as CPN models. By further allowing the shorthand n, where n is any integer, as a shorthand for n‘(), or n tokens with the value (), we can also simulate weighted arcs and how initial markings are typically written in the setting of PT-nets. We also allow inhibitor arcs and reset arcs with the semantics known from PT-net literature [3], i.e., a transition connected to an arc with an inhibitor arc cannot be enabled if there are tokens on the place, and transitions connected to a place with a reset arc will not be inhibited from enabling and upon execution will consume all tokens from the connected place.PT-nets created using CPN Tools can be exported to PNML for analysis in other tools. In the model in Fig. 4, all places but one make use of this shorthand, and transition d has a reset arc and transition c an inhibitor arc. The save file dialog at the bottom-left exposes a save a PNML option.

3.2 Declarative Modeling

Declarative modeling has so far mostly focused on the control-flow perspective, i.e., the order of transitions. We can consider the embedding of declarative languages in CPN Tools as adding constraints to CP-nets or as adding a data perspective to declarative formalisms. From an implementation perspective, we prefer the former. Thus, simulation consists of considering whether a transition instance (or binding element) is enabled in the CP-net sense, and subsequently whether it is also allowed according to the declarative constraint. This prompts an easy means of simulation: we simply run the standard enabling check in CPN Tools (taking data into account) and subsequently (or in parallel) run a declarative check without considering data. In the example in Fig. 4, transitions a, b and d are enabled according to the Petri net semantics, but only a is enabled according to the Declare semantics (the init constraint indicates it must be the first transition to be executed).



File name: model
Save as type: PNML Files (*.pnml)

Fig. 4. CPN Tools 4 with a model mixing Declare, PT-nets, and CP-nets; to the right a model-generated message sequence chart

4 Simulator Extensions

Simulator extensions aim to make it possible to extend CPN Tools. The aim is not to allow external tools to interact with CP-net models, for that Access/CPN is a much better tool, but rather aim to make it possible to make third party extensions of CPN Tools in Java with a look and feel as close as possible to the native feel of CPN Tools. Some features of CPN Tools 4 use extensions for simplified implementation; this includes support for Declare and PNML export.

We have already shown, in Fig. 3, the architecture of CPN Tools with extensions. The extensions can communicate directly with the simulator, and the simulator will take care of mediating communication between extensions and the GUI. To support extensions, we add 6 new kinds of communication, shown in Fig. 5. The first pattern (0) is the old kind of communication used in CPN Tools. The next two patterns augment pattern 0, and allow extensions to inspect and modify communication from the GUI to the simulator (1), and to add new patterns of communication (2). Pattern 3 allows extensions to act like the GUI, and pattern 4 adds a simple remote procedure call (RPC) mechanism making it possible to add inscription extensions implemented in Java. Patterns 5 and 6 add communication to the GUI from the simulator (or extensions), making it possible to create and manipulate graphical elements directly in the GUI.

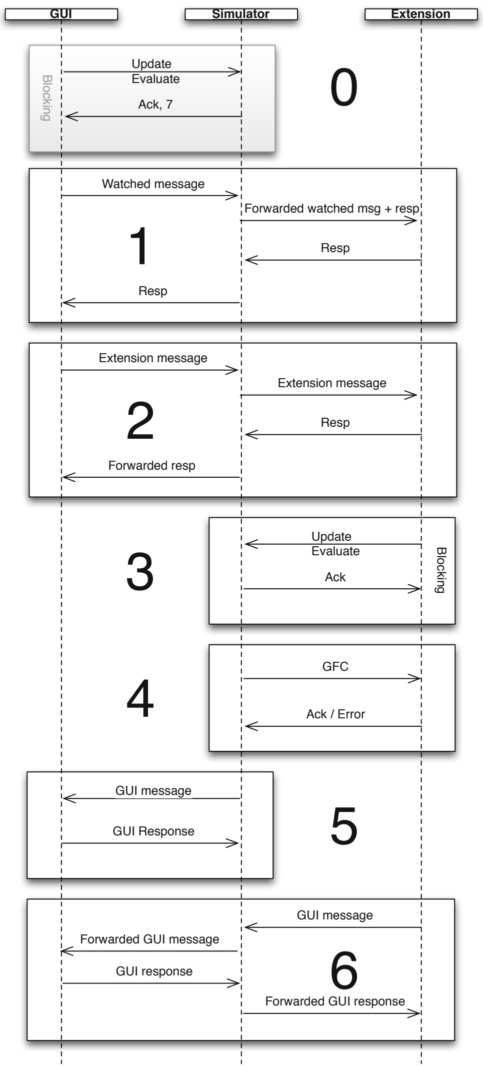


Fig. 5. Patterns of communication allowed for extensions

Pattern 1 informs extensions about the model under construction. It is possible to alter the view the simulator and GUI have of the system. This makes it possible to allow an extension to alter inscriptions, which is for example used to allow time inscriptions to use intervals instead of simple expressions. Filtering the communication from the simulator to the GUI makes it possible to enable and disable transitions, which is used by the extension implementing Declare constraints to disable any transitions not allowed by the constraints.

Some functionality is easier written in Java than in BETA or Standard ML. This may be due to familiarity, a more suitable programing paradigm, or simply due to better library support. For example, PNML export is easily implemented using an XML transformation, but while common in Java, this feature is not available in BETA nor in SML. PNML export in CPN Tools 4 is implemented using a simulator extension written in Java. Similarly, Declare has already been implemented in Java, so using that implementation instead of writing one from scratch makes it much faster to make the implementation and the result is likely to be less error-prone as it has already received testing. This is done with communication pattern 2.

Pattern 3 allows extensions to completely control the simulator. This is for example useful for an extensions exposing a CP-net model as a web-service or by other means let external applications invoke a CP-net model. By allowing generic function calls to extensions using pattern 4, it is possible to expose code in extensions to models. This is used in CPN Tools to expose a Java library for drawing message sequence charts to CPN models.

The GUI callback mechanism (patterns 5 and 6) has existed for some time in CPN Tools, though it has only been used internally by, e.g., Access/CPN 2 to implement cosimulation [13]. By extending this mechanism, it is now possible to directly invoke code in CPN Tools from extensions. Most importantly, it is possible to create pages and add graphical elements to them. This makes it possible to implement visualizations directly in the CPN Tools GUI. One such extension draws message sequence charts in the CPN Tools GU with the layout logic written in Java.

Often, extensions need several of the communication patterns; for example, the Declare extension adds new commands for syntax checking Declare constraints (pattern 2) and filters communication from the simulator to disable transitions (pattern 1). The message sequence chart extension asks the simulator to instantiate stub-functions for drawing message sequence charts (pattern 3). These functions call into the extension (pattern 4), which makes callbacks to

Listing 1. Parts of Declare extension

```
public class DeclareExtension
    extends AbstractExtension {
  static final int ID = 10001;
  Option<Boolean> SMART =
    Option.create("Smart", "smart",
            Boolean.class);

  public DeclareExtension() {
    addOption(SMART)
    addSubscription(
        new Command(400, 2)); }

  public int getIdentifier() {
    return ID; }
  public Strig getName() {
    return "Declare"; }

  public Packet handle(Packet p) {
    switch (p.getInteger()) {
      case 400: ...
      ...
      case 10001: ... } }
  ... }
```

Listing 2. Parts of MSC extension

```
public class MSCExtension
    extends AbstractExtension {
  public Object getRPCHandler() {
    return new Dispatcher(channel); }
  ... }

public class Dispatcher {
  Channel c;
  int serial = 0;
  Map<Integer, Canvas> mscs = ...
  public Dispatcher(Channel c) {
    this.c = c; }

  public Integer createMSC(String n) {
    int id = serial++;
    mscs.put(id, new Canvas(c, n));
    return id; }

  public void addEllipse(Integer id) {
    Canvas cv = mscs.get(id);
    cv.add(new Ellipse(10, 10, 60, 40)
        .setBackground(Color.GREEN)); }
  ... }
```

the GUI for performing the actual drawing inside the CPN Tools GUI (pattern 6). The chart at the right of Fig. 4 is created using this extension, and illustrates (simplified) the communication taking place to draw itself. In Listings 1 and 2, we see fragments of the implementations of extensions. Extensions must define a name (ll. 15–16) and a numerical identifier (ll. 3, 13–14) so the extension manager can tell all running extensions apart. Extensions can have options (ll. 4–6, 9) which are exposed in the GUI and automatically transmitted to the extension when changed. Handling packages received for communication patterns 1 and 2 is done by implementing a handle method (ll. 18–22). An extension says it wants to intercept packages using pattern 1 by subscribing to them (ll. 10–11). Handling pattern 4 is done by returning a RPC handler (ll. 33–34); all methods are automatically made available in the simulator. All communication (using pattern 6) for drawing is abstracted away by object oriented primitives (ll. 44–52).

5 Conclusion and Future Work

CPN Tools 4 improves on an already useful tool in two main areas: providing end-users with conveniences making them more efficient, and providing developers with an extension mechanism that can be used to make extensions of CPN Tools that feel close to native.

CPN Tools 4 provides syntactical sugar making it very easy to make Place/-Transition Petri net models. This is performed as a backward- and forward-compatible embedding of the formalism in CP-nets. In addition, CPN Tools adds support for common low-level special arcs, including inhibitor and reset arcs, and allows saving models using low-level constructs only in the PNML format. CPN Tools 4 also adds support for the Declare language, which allows modelers to focus less on the actual execution order but instead on constraints on the order of execution. Some of the user-facing features could not have been developed without the use of the other major new feature of CPN Tools, simulator extensions. This feature makes it easy to extend CPN Tools using Java code. Several extensions ship directly with CPN Tools; some features of CPN Tools appear as completely native features, but are realized using extensions, including Declare support, PNML export, and drawing message sequence charts from model executions. The default distribution includes a scene-graph-based library for maintaining visualizations in the CPN Tools GUI without worrying about the underlying protocol.

While we have extended CPN Tools to make it easy to make different kinds of models, we do not aim for a generic framework for Petri nets like the Petri Net Kernel [7]. We still deal primarily with CP-nets, and the embedding of PT-nets is just that, an embedding, which means we do not do any of the advanced analysis facilitated by looking at low-level nets, such as advanced state-space analysis as performed by LoLA [18], stochastic/timed analysis as performed by GreatSPN [5], or symmetry reduction as performed by CPN-AMI [11]. Adding Declare constraints is interesting as it provides Declare models with data (from the CP-nets) and provides CP-nets with a new description of control-flow at at higher level, making it easier to step-wise refinement by focusing on abstract control flow first, and add concrete control flow and data as necessary.

CPN Tools is available free of charge from `cpntools.org`; the GUI runs on Windows, and the simulator and extension manager on Windows, Mac OS X, and Linux.

References

1. van der Aalst, W., Stahl, C., Westergaard, M.: Strategies for Modeling Complex Processes using Colored Petri Nets. Transactions on Petri Nets and Other Models of Concurrency (to appear, 2013)
2. CPN Tools webpage, `http://cpntools.org`
3. Desel, J., Reisig, W.: Place/Transition Petri Nets. In: Reisig, W., Rozenberg, G. (eds.) APN 1998. LNCS, vol. 1491, pp. 122–173. Springer, Heidelberg (1998)
4. Gallasch, G., Kristensen, L.: A Communication Infrastructure for External Communication with Design/CPN. In: Proc. of Third CPN Workshop. DAIMI-PB, vol. 554, pp. 79–93 (2001)
5. GreatSPN webpage, `http://gwww.di.unito.it/~greatspn/`
6. Jensen, K., Kristensen, L.: Coloured Petri Nets – Modelling and Validation of Concurrent Systems. Springer (2009)
7. Kindler, E.: The ePNK: An Extensible Petri Net Tool for PNML. In: Kristensen, L.M., Petrucci, L. (eds.) PETRI NETS 2011. LNCS, vol. 6709, pp. 318–327. Springer, Heidelberg (2011), `gdx.doi.org/10.1007/978-3-642-21834-7_18`
8. Kristensen, L., Mechlenborg, P., Zhang, L., Mitchell, B., Gallasch, G.: Model-based Development of a Course of Action Scheduling Tool. STTT 10(1), 5–14 (2007)
9. Lindstrøm, B., Wagenhals, L.: Operational Planning using Web-Based Interfaces to a Coloured Petri Net Simulator of Influence Nets. In: Proc. of FMADS. CRPIT, vol. 12, pp. 115–124 (2002)
10. Rasmussen, J., Singh, M.: Mimic/CPN. A Graphical Simulation Utility for Design/CPN. User's Manual, `http://www.daimi.au.dk/designCPN`
11. The MARS Team: CPN-AMI webpage, `http://www-src.lip6.fr/logiciels/mars/CPNAMI/`
12. Westergaard, M.: The BRITNeY Suite: A Platform for Experiments. In: Proc. of 7th CPN Workshop. DAIMI-PB, vol. 579, pp. 97–116 (2006)
13. Westergaard, M.: Access/CPN 2.0: A High-level Interface to Coloured Petri Net Models. In: Kristensen, L.M., Petrucci, L. (eds.) PETRI NETS 2011. LNCS, vol. 6709, pp. 328–337. Springer, Heidelberg (2011)
14. Westergaard, M., Evangelista, S., Kristensen, L.M.: ASAP: An Extensible Platform for State Space Analysis. In: Franceschinis, G., Wolf, K. (eds.) PETRI NETS 2009. LNCS, vol. 5606, pp. 303–312. Springer, Heidelberg (2009)
15. Westergaard, M., Kristensen, L.: The Access/CPN Framework: A Tool for Interacting With the CPN Tools Simulator. In: Franceschinis, G., Wolf, K. (eds.) PETRI NETS 2009. LNCS, vol. 5606, pp. 313–322. Springer, Heidelberg (2009)
16. Westergaard, M., Lassen, K.: The BRITNeY Suite Animation Tool. In: Donatelli, S., Thiagarajan, P.S. (eds.) ICATPN 2006. LNCS, vol. 4024, pp. 431–440. Springer, Heidelberg (2006)
17. Westergaard, M., Maggi, F.: Declare: A Tool Suite for Declarative Workflow Modeling and Enactment. In: Proc. of BPMDemos. CEUR Workshop Proceedings, vol. 820. CEUR-WS.org (2011)
18. Wolf, K.: Generating Petri Net State Spaces. In: Kleijn, J., Yakovlev, A. (eds.) ICATPN 2007. LNCS, vol. 4546, pp. 29–42. Springer, Heidelberg (2007)