# Contents

**From the Editors** ................................................................. 1

**Imprint** .............................................................................. 2

**Cover Picture Story**

*Andreas Eckleder, Thomas Freytag:*
WoPeD – A tool for teaching, analyzing and visualizing workflow nets ............................................ 3

**Tool Paper Guideline**

*Giuliana Franceschinis, Kees van Hee, Ekkart Kindler, Fabrice Kordon, Lars M. Kristensen, and Karsten Wolf:*
Proposal for: Requirements and Evaluation of tool papers for PETRI NETS ........................................ 9

**Technical Contributions**

*Jean-Baptiste Voron, Fabrice Kordon:*
Evinrude: A Tool to Automatically Transform Program’s Sources into Petri Nets .............................. 19

*Franck Pommereau:*
Quickly prototyping Petri nets tools with SNAKES ............................................................................. 39

**Order Form** ........................................................................ 57

---

Deadline for next volume: March 15, 2009
From the editors

This issue of the Petri Net Newsletter is again devoted to tools. As announced in the last issue, it contains two more refereed technical contributions which are extended versions of contributions to the international Workshop on Petri Net Tools and Applications (PNTAP) 2008. The first article, by Jean-Baptiste Voron and Fabrice Kordon, describes the tool Evinrude for the translation of C source code into Petri nets for model checking purposes. The second article, by Franck Pommereau, presents the toolkit SNAKES for quickly prototyping Petri nets tools.

The cover picture story presents the open source tool WoPeD (Workflow Petrinet Designer), maintained by the group of Thomas Freytag, giving teachers and researchers a powerful and flexible e-learning and e-publishing instrument.

Moreover, this issue contains a guideline for the publication of papers on software tools, in particular for Petri net tools, illustrated by an example. The need for such a guideline was discussed at several program committee meetings of the Petri Net Conference. It is written by Giuliana Franceschinis, Kees van Hee, Ekkart Kindler, Fabrice Kordon, Lars M. Kristensen and Karsten Wolf, and should help authors in writing (better) Petri net tool papers.

There were several very good presentations at the AWPN 2008 workshop in Rostock. Therefore we decided to include extended contributions into the next issue in April 2009. Nevertheless the reader should feel free to additionally submit technical contributions or work in progress to the April 2009 issue, since there is place for up to five papers.

Unfortunately, in this issue we are forced to omit the bibliography with the latest published papers on Petri nets. Since this time there are twice as much new papers as usual, we would not be able to cover the printing costs if the bibliography were included. Still, the bibliography is available online. It will be discussed by the editorial board how to proceed in future.

On behalf of all editors,
Augsburg, October 2008
Robert Lorenz
The Petri Net Newsletter is an information service of the Special Interest Group FG 0.0.1 “Petri Nets and Related System Models” of the Gesellschaft für Informatik (GI), Bonn, Germany. Two issues are being published per year. The editorial board of the Petri Net Newsletter is formed by:

Jörg Desel
Ekkart Kindler
Kurt Lautenbach
Robert Lorenz
Daniel Moldt
Rüdiger Valk
Karsten Wolf

Scope: The Petri Net Newsletter serves as a medium for the rapid distribution of any information about Petri Nets and related system models all over the world. Topics include:

- reviewed technical contributions including surveys and state-of-the-art-reports
- work in progress including problems and puzzles
- reports on departments, institutes, companies, projects, local activities
- information about new books and PhD thesis
- abstracts of recent publications

Subscription: Members of FG 0.0.1 of the GI receive one copy of the Newsletter free of charge. On the last page of this issue, you will find an application form containing the address of the GI. Additional and former issues can also be obtained from the GI.

Contributions: Any contributions to the field are welcome. It should be clearly stated whether a contribution is submitted as a technical contribution or as work in progress. Papers submitted as technical contributions will be reviewed. They should be 6 to 15 pages long and in A4 format. Articles submitted as work in progress will not be reviewed. As contributions will be printed as submitted, make sure that no space is wasted. Contributions should be sent to:

Ekkart Kindler
Informatics and Mathematical Modelling
Technical University of Denmark (DTU)
Petersens Plads
Kgs. Lyngby
Denmark
eki@imm.dtu.dk

Robert Lorenz
Lehrprofessur für Informatik
Universität Augsburg
Eichleitnerstr. 30
86159 Augsburg
Germany
robert.lorenz@informatik.uni-augsburg.de

Conference Announcements and Conference Reports: Conference announcements such as Calls for Papers and Calls for Registration are no longer in the focus of the Petri Net Newsletter. On special request, they will still be published. They should be formatted according to the layout of the Newsletter and take at most 2 pages.

Recent Publications: In the newsletter, the new entries of the online Petri Net Bibliography (see http://www.informatik.uni-hamburg.de/TGI/pnbib/) are printed. The bibliography is maintained by Frank Heitmann. To keep the bibliography as complete as possible, please send new bibliography entries by e-mail to: heitmann@informatik.uni-hamburg.de (Frank Heitmann) containing name of author(s), title, book, ..., and abstract.

Deadlines: There are two issues per year. Deadlines are the mid of March and the mid of September.
WoPeD –
A tool for teaching, analyzing and visualizing workflow nets

Andreas Eckleder¹, Thomas Freytag²

¹ Nero AG, Karlsbad, Germany
eckleder@woped.org
² University of Cooperative Education (Berufsakademie), Karlsruhe, Germany
freytag@woped.org

Abstract. WoPeD (Workflow Petrinet Designer) is a Java-based, open-source software tool developed at the University of Cooperative Education ("Berufsakademie") Karlsruhe, Germany. WoPeD uses the Petri Net Markup Language (PNML) as interchange format and is able to edit, simulate and analyze plain P/T nets and in particular "van-der-Aalst"-like workflow nets. The main focus of WoPeD lies on simplicity, interactivity and visual expressiveness, giving teachers and researchers a powerful and flexible e-learning and e-publishing instrument.

Introduction

WoPeD is an easy-to-use software tool for editing, simulating and analyzing workflow nets as well as plain place-transition Petri nets. WoPeD is being developed continuously by faculty members and students at the University of Cooperative Education ("Berufsakademie") Karlsruhe/Germany in conjunction with a group of graduates who now are IT professionals in local partner companies. WoPeD is published under the LGPL, an open-source license which allows free access to its source code and underlying design documents. The Java-based tool runs on all operating systems for which Java version 5.0 or higher is available - in particular on all Windows platforms, most Linux distributions and MacOS X.

WoPeD primarily focuses on educational and publishing purposes, supporting lecturers, researchers and students in the area of Petri nets and workflow management. WoPeD is relating strictly to the well-established "van-der-Aalst" notation [Aal02] and can visualize both structure and dynamics of workflow nets, providing a deeper understanding of the underlying principles, notational concepts, key properties and algorithms.

By this, WoPeD can be considered as an e-learning or e-publishing instrument. It is a supporting tool for creating lecture materials or setting up and performing student exercises, assignments and case studies. Additionally, it can be used to illustrate scientific publications through tailored examples or other visualizations. A lot of positive feedback from all over the world has shown that WoPeD has successfully been applied in all of these areas.

A couple of publications have accompanied the emerging development of the software, giving additional information on the underlying architecture [FrL03], on the used algorithms [Eck06], on some experimental visualization concepts [FlF05] and on new functional features [EcF08].

The development of WoPeD started in May 2003. The first public beta was brought-up in March 2005, since then several major releases have been made available. The current release 1.6 was provided in April 2008, the next major release 2.0 will be published by the end of 2008. Source code and installer packages are hosted on Sourceforge, a well-established development platform for open-source projects. Via Eclipse, the whole source code (all releases since 2005, including the current development branch) can be accessed in a straightforward way by anonymous synchronisation with the CVS repository.

WoPeD's architecture is based strictly on the Model-View-Controller (MVC) pattern. The data model is represented by standard PNML, the code to export to and import from the underlying file system is automatically generated from the XML schema. The graphical view is based on a Petri-net-specific implementation of the well-known JGraph framework [JGr08], a widely-used open-source-solution for drawing, transforming and visualizing graphs.
Main features

The following section gives a brief overview on the most important features of WoPeD, especially in the light of using the tool in the area of academic teaching as a "blended learning" instrument.

Graphical process editor with sub-process support

The major development goals of WoPeD are simplicity, lean architecture and intuitive usability. The graphical net editor is not only capable to represent traditional place-transition Petri nets, but also offers support for the class of workflow nets including special operators like XOR-split/join, AND-split/join, trigger symbols and resource assignments. It also provides functions to define hierarchical sub-processes as annotated transitions which act as placeholders for an embedded workflow net. By this, even huge process models can be represented graphically and analyzed by applying the associated algorithms only on the local process level (e.g. soundness check, reachability/coverability graph construction).

![Fig. 1: A workflow net with a sub-process and process structure sidebar](image)

PNML-compliant file format and multiple export formats

The standard file format of WoPeD is PNML [WeK03], allowing export or import of net models to and from other (PNML-based) Petri net tools. In order to guarantee a maximum of compatibility, the workflow net control flow extensions are internally translated into plain Petri net constructs and thus can be loaded also by tools unaware of the workflow-specific extensions. Additionally, WoPeD provides several export interfaces, including the JPEG, GIF and PNG graphics formats and textual formats like TPN (for Woflan interaction) and BPEL (see below).

Resource view editor and resource mapping

WoPeD contains a separate graphical resource modelling component to create and manage resource classes (i.e. groups and roles) and their contained resource objects (i.e. workflow participants) for each modelled process. Inside the process editor, each resource-triggered transition can be associated...
with a role and a group. The notation used for this also relates almost completely to the "van-der-Aalst/van Hee"-suggestions [AaH02]. By following this graphical approach, students intuitively get an idea on how the execution conditions of process models are connected to their organizational context.

**Interactive token game**

WoPeD provides a graphically animated token game simulator which can be controlled through a "remote-control"-like interface. It is possible to step forward and backward in single or in multiple steps, create a log of the chosen firing sequence and restore it. When encountering a sub-process transition, the user can choose between stepping over or stepping into the sub-process, very much comparable to graphical debugging in software development tools. The comfortable and highly intuitive interface helps students to get a deep understanding of process behaviour and how it is semantically represented by the Petri net model and the firing rule.

![WoPeD Graphical Token Game](image)

**Qualitative analysis**

WoPeD can investigate a variety of qualitative properties, e.g. free-choice situations, S-component coverage, well-structuredness and soundness. Most of these properties are checked by built-in algorithms, except some runtime-critical parts of the soundness check which are computed by using an interface to the (non-graphical) Woflan tool [Wof08] which is developed at the TU Eindhoven. This "WoPeD-Woflan"-bridge allows the direct representation of the analysis results inside the associated workflow net graph and its nodes, visualizing control flow errors in process models and possible reasons for it on a graphical base. The process editor forces all sub-processes to be workflow nets by themselves, and thus restricting them to have exactly one input and one output place. This has the

1 The "WoPeD-Woflan" bridge is currently available for Microsoft Windows® operating systems only
interesting consequence that dynamical properties including soundness can be checked locally and separately for each sub-process.

### Quantitative analysis and capacity planning

WoPeD is able to store and visualize an average service time value with each resource-trigged transition and an average branching probability with each outgoing arc of an implicit or explicit XOR-split operator. This allows the computation of capacity planning scenarios derived from both process and resource models, assigning each transition the expected number of work items per case, and each resource class the minimum required number of members under a given resource utilization rate. The algorithm to compute the number of work items per case is based on a finite prefix of the net unfolding being capable to handle possibly infinite loops. Apart from this, WoPeD contains a discrete event simulator which is able to compute quantitative results like average waiting time and resource usage based on random number generators with various distribution functions. The simulations use the underlying process and resource model and can be configured by a variety of parameters such as average arrival rate of cases, average service time of transitions and average branching probability of XOR-splits.

### Reachability and coverability graph visualization

The new release 2.0 of WoPeD implements automatic reachability and coverability graph construction and visualization. The current implementation provides two simple layout mechanisms for the visual representation, leaving the manual "fine-tuning" of node positions to the user. Once displayed in a satisfying way, the coverability graph can also be exported to the most common graphics formats such
as JPEG, BMP and PNG. By this, WoPeD can be used to create sample graphs e.g. for lecture material or other publications.

![Image of a coverability tree construction and visualization](image)

**Fig. 4: Coverability tree construction and visualization**

As the only known Petri net tool, WoPeD is able not only to construct behavioural information in memory, but also to visualize the associated graph on the screen. By this, students learn about the impact of their modelling decisions on the associated state space and get a good visual impression of Petri net behaviour in general and the "state explosion" phenomenon in particular.

### BPEL export

An important new feature of WoPeD 2.0 is the export of well-structured, free-choice workflow nets into the widely-used BPEL format. The process control flow elements are converted into the associated BPEL constructs and single transitions can be used as placeholders for basic BPEL operations (*assign, invoke, receive, reply, wait*). A global namespace is supported for defining state variables which can be used as parameters when interacting with web-services. By this, WoPeD allows the orchestration of arbitrary web services identified by partner links entered manually or imported from UDDI business registries. The parser used to convert the workflow net control flow into an executable BPEL script is based on the ideas published in [AaL08] and [Las06].
Conclusion

WoPeD is an evolving software tool with lots of useful functional features. The editing component supports a process model view as well as a resource model view. WoPeD fully supports the original workflow net notation and contains algorithms for checking qualitative properties (soundness) as well as quantitative properties (capacity planning). With its high degree of intuitiveness and its easy-to-use interface, WoPeD is an ideal instrument for "blended learning" in the context of teaching and publishing in the area of workflow management and process analysis. For a closer look at the tool's functionality, including download links, screenshots and documentation, please refer to the website [WoP08].

References


Quick facts (from Petrinet tools database)

Homepage   www.woped.org
Email contact info@woped.org
Availability Free (open-source)
Platforms   Independent (Java)
            One-file-installers for Windows, Linux and MacOS
Net class   Workflow nets, place-transition nets
File formats PNML, TPN (export), JPEG (export), GIF (export), PNG (export), BPEL (export)
Features   Graphical editor, sub-processes, interactive simulator, qualitative analysis, soundness check, capacity planning, quantitative simulation
Proposal for: Requirements and Evaluation of tool papers for PETRI NETS

Giuliana Franceschinis, Kees van Hee, Ekkart Kindler, Fabrice Kordon, Lars M. Kristensen, and Karsten Wolf

Abstract. This paper gives guidelines for the publication of papers on software tools, in particular for Petri net tools. The guidelines are illustrated by an example.

1 Introduction

In computer science in general and in the field of Petri nets in particular, software tools are becoming more and more important. Often, the scientific work in our field results in algorithms and techniques that can be supported by software tools, which can be applied either by researchers or by practitioners to analyze or design complex systems. Society is more and more expecting practical results from scientists. Hence, in many cases, stand-alone scientific publications will not be enough in the future.

For these reasons the PETRI NETS commnity (officially: International Conference on Application and Theory of Petri Nets) stimulates the development of software tools by giving the opportunity to researchers to publish tool papers in the proceedings of the PETRI NETS conferences. Papers can be submitted in a special category tool paper to PETRI NETS.

In the last years, there have been several problems with submitted tool papers, and there has been a recurrent discussion on how a good tool paper should look like. Some tool papers seemed to present a nice tool, but the presentation was not really helpful to the audience of PETRI NETS. Other tool papers looked very much like regular papers presenting some algorithm or method; but the authors, for whichever reason, submitted them as tool papers. Part of the problem was, that there are no clear guidelines what a tool paper should cover and how a good tool paper should look like. This paper should give some guidelines for writing as well as for evaluating tool papers at PETRI NETS.

2 Requirements

This section states some basic requirement for tool papers and the presented tools. Clearly, the focus of a tool paper should be on the tool and present the tool and its features, and the features should be presented from a user’s point of view. It should not focus on the inner workings of the tool, or on some algorithms and methods that could be published as a regular paper.
Therefore, the most basic acceptance criterion is that the presented tool is available for evaluation for the reviewers during the reviewing process and, if the paper is accepted, for the readers. This does not require that the tool is open or free software; but it requires that at least a demo version with some documentation is freely available, preferably via a web-page. The demo version must make it possible to evaluate the features of the tool that are presented in the paper.

Moreover, a tool paper must present a tool that has not been presented before as a tool paper in the proceedings of PETRI NETS. And the tool must be of sufficient interest for the community; this could be either for the methods or algorithms it features, for its advanced usability or smoother integration of different methods, for a new open architecture, or for the general potential of being a widely used tool. In exceptional cases, a tool that was presented at PETRI NETS already can be presented another time. In such cases, the tool must have significant new features, and the paper must clearly point out the new features that justify another presentation of that tool. Note that a tool can be a monolith, which means that it is one piece of software with a clear functionality or it can be a set of related tools, under one name. In the latter case we consider each member of the set as a tool itself.

3 Format

A tool paper should clearly address the following issues:

1. The objectives of the software tool: A description of the purpose and application domain of the tool, its intended use and application scenarios, as well as the typical types of users.

2. The functionality of the tool: What can you do with the tool and what is the input and resulting output (at an abstract level). This can be done by means of use cases, preferably using a (small) running example. Here also the essential methods and algorithms that are used to transform input into output could be explained, again, at a high level. Note that the paper should not present the theory behind the tool, but instead should provide references to the literature.

3. The architecture of the software tool: How is the tool composed from existing or new components and what are the main design decisions and rationale underlying the tool? For each relevant new component the functionality should be described and the interfaces with other components. Further, if relevant, the information architecture (logical structure of the database of the tool) should be described. Finally, the runtime environment of the system should be specified. Moreover, the interfaces to other tools and possible interchange of data with other tools should be discussed.

4. Some interesting use cases that illustrate the working and use of the tool and, possibly, a discussion on the experiences obtained while using the tool. A detailed evaluation of a tool, however, is beyond the scope of a tool paper; this could be a regular paper. If such studies exist already, a tool paper is free to refer to them.
5. A comparison with other tools or a former version of the same tool. A tool paper should clearly point out what its main characteristics are and relate them to the features and characteristics of existing tools. Also, a discussion of the limitations of the tool is welcome.

6. The paper should clearly indicate where the information can be found to obtain, install, and start the tool, and what the license conditions are. This information, possibly available on the web, must provide enough details so that the intended type of user is able to get the tool started.

Note that a tool paper is limited to 10 pages in LNCS format. Therefore, each of these issues must be described in a compact way. For more detailed information, the paper can refer to other publications and, preferably, some web pages. Still a tool paper must be readable in isolation, i.e. it should be self-contained.

4 Example

There are many good examples of tool papers of PETRI NETS (see e.g. the first two references). Here, we use the first one as an illustration of these guideline: “Petriweb: a Repository for Petri Nets [1]”. We made some changes to the original paper in order to better illustrate these guidelines.

4.1 Objectives

Many tools exist to support the modeling process. One tool may be better suited for designing models, another for analysis. Therefore, in a typical development or research environment, multiple tools are used in combination. This creates the need to work on the same models with different tools. This can be addressed by defining a standard file format that all tools can use.

Models need not only be shared by tools, but also by different users. For instance, a model may need to be reviewed by a colleague of the designer. We can send the model, and the colleague can open and use the model, but as soon as new versions appear, it becomes hard to make sure the right version is always used. Here, a shared location for the models is needed.

A shared collection of models also encourages users to reused existing models or parts of them. This can be useful for different kinds of users. Designers, who employ modeling to describe and design systems, can use this to streamline the modeling process. Researchers and educators can build up collections of models used as illustrations, e.g. examples or counterexamples in proofs.

Most collections of models will be assembled in the context of a specific project, with a small group of participants. But collections can also be turned into company-wide or world-wide resources. In such cases, users will rarely be familiar with all the models in the collection, and collections can grow quite large.
The tool presented here is meant for the management of large collections of models, in particular for storage and retrieval of models and for interfacing with tools to manipulate the models.

A specific domain of application is process modeling with Petri nets at our university. Many examples in course material and exercises are reused over the years; they are often recreated from memory or from paper. It is attractive to make such examples available in a shared repository, accessible by both students and teachers. Here, the need for both browsing and searching facilities is evident. Since users recognize Petri nets by their graphical representation, browsing the collection can only be supported with a graphical browser. In larger collections, users also need to filter the collection based on properties of the content. For instance, users may want to find examples of Petri nets that are bounded and contain a deadlock.

4.2 Functionality

Petriweb is a web application for managing repositories of Petri nets. A Petriweb installation can host many different collections, each with their own administrators and users. Petriweb has two sets of functions:

– Retrieval of Petri net models.
– Model management.

An example query Petriweb needs to support: “Find the smallest net present that is live, unbounded and free-choice”. Another example, from a developer’s perspective: “Find a component with this interface and performing task $P$”. Queries must also include metadata, e.g. the author or original publication of the net. We see that Petriweb must support different kinds of properties: structural properties, e.g. the number of places; behavioral properties, such as boundedness or liveness; and metadata. Retrieval is performed by means of the following search criteria.

– Metadata
  By allowing metadata as properties in Petriweb, the user uploading the net can specify related information about it, such as when it was created, by whom, etc.

– Application characteristics
  They describe what the model is about: in what domain, for what purposes, etc.

– Structural and behavioral properties
  Simple structural properties can be determined by simple programs, while more complex structural and behavioral properties such as free choice, liveness, boundedness, can be determined with existing Petri net analysis tools. Many such tools can be called as filters that take a net as input and produce results as output. Petriweb can incorporate this through automated properties. These are not specified by the user while uploading the net, but instead, computed automatically by invoking an XSLT stylesheet or an external command. This allows non-trivial structural and behavioral properties to be determined automatically; they can even be used in search criteria.
Transitions

This mechanism can also be used to automate conversions from PNML to other file formats, e.g., the TPN format of Woflan [2]. Calling an analysis tool is often preceded by a transformation, but the results can also be presented to the user, e.g., as the input for a client-side tool. In supporting transformations, Petriweb becomes more than a repository: it functions as a mediator between different tools and formats.

Fig. 1 shows the user interface of Petriweb to enter the search criteria. Not only searching on properties is important, the graphical representation of a Petri net is also a great help in finding a specific Petri net. It is hard to describe a Petri net in words, such that others can find it; the graphical display of a net is much easier to recognize. Therefore, search results do not only list the Petri nets’ names and properties, but also display them as diagrams. This allows a combination of searching and browsing. Fig 2 shows the user interface of Petriweb for browsing the search results.

The second group of functions concerns model management.

Addition

Petriweb has been designed for public and private use. Petri nets can be shared within a community. Anyone can register at Petriweb and upload their own Petri nets. To allow this, a standard file format is used for uploading Petri nets: the Petri Net Markup Language (PNML). Before adding the Petri net, Petriweb first checks its syntax against a PNML syntax definition [3]. The net is then parsed and stored.
– Approval
While anyone can be registered at Petriweb and upload Petri nets to it, nets go through a built-in approval mechanism. A net can be in three possible states: uploaded, approved, or deleted. After a user uploads a Petri net for a community, the community moderator receives a notification and can decide whether the Petri net is approved or denied. In this way, collections remain manageable.

– Retrieval
Sharing the Petri nets also means that it is possible to download and use the Petri nets from Petriweb. Petriweb supports this in two ways, by allowing downloading the original uploaded file, or a file generated from the parsed information. In this way it is possible to serve as many tools as possible.

Petriweb is primarily intended to serve the public community by sharing Petri nets. Users may not want to share their Petri nets with the total public community, but only with a small group of users. Therefore, Petriweb features a built-in mechanism to support communities. Each registered user can request a community. After the Petriweb administrator has approved the request, the user becomes moderator of the private community and can invite users to join. In this way restricted project areas can be defined and used, with the full search and properties functionality of Petriweb.
Fig. 3. Data schema of Petriweb
4.3 Architecture

Petriweb is based on the data schema given in Fig. 3. It consists of three parts:
– the structure and marking of the Petri net,
– the graphical information to display the Petri net, and
– the properties associated with the Petri net.

Petriweb supports component based models: definitions define the behavior and structure, and can be instantiated in other definitions. Each Petri net definition is either a transition definition or a (sub)net definition. For every transition and subnet, its definition and its instance are stored. Every instance is a part of a (sub)net. The relation between ProcessInstance and Subnet depicts the hierarchy of the Petri net. Places are also part of a (sub)net. A marking is stored in the MarkingInformation table. A Petri net can have a trace of markings (a firing sequence); this information is stored in the StateHistory table. Multiple markings and traces per net are supported. Arcs are not considered as separate objects in the Petri net, but instead, both transitions and subnets have connectable pins. These pins connect instances with places or other pins. Due to the separation of definition and instance, a distinction is made between formal and actual pins. A formal pin is part of the definition, an actual pin is an instance of the formal pin. Graphical information is part of the data schema, although not all relations are drawn in the figure.

Properties Since different uses of the repository gives need of different kind of properties, Petriweb is designed to give each community or installation its own properties, i.e. the moderator of each community or installation needs to define the properties which have to be filled in when uploading a Petri net. In this way properties are as generic as possible. In Petriweb properties are stored in the table Properties. Each property belongs to a specific category. The table PropCategory contains information about the different categories. The table NetProperty contains the values of properties of (sub)nets in the repository. The table contains some standard properties that should always be available for every Petri net, such as its name, the location of its file and whether the (sub)net is approved. For each property defined in Petriweb (thus an entry in the table Properties) a column is present in this table. Automated properties can be derived either by applying a stylesheet (XSLT) or by calling a tool on the command line. The output is parsed into the database. In this way, any tool can be used to generate the value of a property, hence supporting as many disciplines as possible.

Communities To support communities, it is of importance to separate the different communities from each other, in such a way that there is no connection between them. An advantage of this approach is that only properties needed in the community are stored and shown. To support this, each community receives its own database to fill. In order to administer the different communities, the data schema of Fig. 4 is used.
Design The design of Petriweb is component based (Fig. 5). It is divided into three main components: repository, viewer and checker. The repository uses the viewer and checker. All components are designed and implemented as stand-alone applications. It was a goal to provide easily installation on different machines.\textsuperscript{1} The implementation in PHP with MySQL meets this goal. Properties are designed in such a way that third party tools can be part of the repository. The implementation is such that it is also possible to integrate the upload and search of Petri nets into stand-alone tools, like an editor or analysis tool. The software is portable; different Petriweb installations can be easily installed. A publicly available installation of Petriweb can be found at http://www.petriweb.org. Also the source code is publicly available, and can be downloaded from the public website.

4.4 Use case

Typical uses of Petriweb is to search for Petri nets with certain properties, to verify a hypothesis, or disproof a property. See Fig. 2 and 1. The found Petri nets can be tested on the hypothesis. If all Petri nets satisfy it, one has to search for a counter example by hand. If such a counter example is found, the user can upload it, and specify meta data of the Petri net. Petriweb then calculates the automated properties. After the moderator approves the Petri net, it is added to Petriweb, and retrievable for other users.

\textsuperscript{1} Petriweb has been tested on two platforms: Linux and Microsoft Windows.
4.5 Comparison with other tools

There are various solutions for sharing data. One of them is to use standard version control software, such as CVS or Subversion. However, these systems still require the users to be familiar with the organization of the material in terms of file names and directory structure. For larger collections, or an open-ended user base, this does not suffice. Additional facilities for searching and browsing will be required that employ knowledge of the model contents. Therefore, specialized repository software is needed that combines general file management with domain specific knowledge.

The only tool we know of that has comparable functionality is the workflow patterns website (cf. www.workflowpatterns.com). The difference is that this site is only meant for retrieval of patterns and not for mediator type functions of Petri web. Hence, in the workflow patterns website is it not possible to upload patterns, to share patterns in a closed community or to retrieve patterns based on some property.

4.6 Installation

Petriweb can be used by logging in as user op www.petriweb.org. There the user can create his own collection of models and he can use the existing set of public models. If one wants to obtain the sources please contact info@petriweb.org.

References

Evinrude: A Tool to Automatically Transform Program’s Sources into Petri Nets

Jean-Baptiste Voron & Fabrice Kordon

Université Pierre & Marie Curie
UMR CNRS 7606, LIP6 MoVe
4, place Jussieu, Paris, F-75005 France
jean-baptiste.voron@lip6.fr & fabrice.kordon@lip6.fr

Abstract. Model checking is a suitable formal technique to analyze parallel programs’ execution in an industrial context because automated tools can be designed and operated with very limited knowledge of the underlying techniques. However, the specification must be given using dedicated notations that are not always familiar to engineers (so far, model checking on UML raises complex problems that will not be solved immediately).

This paper proposes an approach and its implementation as a tool to perform transformation of C source code into Petri nets, a suitable specification for model checking. To overcome the complexity of the resulting specification, we focus on specific aspects of the program. Hence, we never model the entire processed program, but only its relevant parts.

In this paper, we will apply this approach on some examples using our tool: Evinrude.

1 Introduction

Behavioral analysis of concurrent systems cannot be completed anymore using only “traditional” test-based approaches. First, their complexity often makes impossible to cover a significant part of the state space by simulation. Second, testing concurrent systems is not trivial and may lead to complex problems like probe effects [1]. To overcome these limitations, it is now recognized for many years that formal methods are of interest since they provide more trustworthy and mathematically founded information [2, 3].

Among the available formal verification techniques, model checking is particularly interesting due to its potential for full automation as well as its error reporting capabilities [4, 5]. So, neither a long training nor a long practice are required from engineers, using model checkers to extract execution paths leading to undesired behaviors.

However, the problem is about the way engineers design specifications too. Most model checkers require formal specifications as inputs: Automata [6], Promela [7], Petri nets [8], etc. These input formalisms may be difficult to learn. They usually also propose a low level of abstraction not adapted to their use in the context of industrial-size projects without extensive practice.
A way to avoid the learning of such languages is to consider verification at source program level. This cannot replace a modeling/verification phase but it is a way to tackle the specification problem that is crucial, especially in domains related to security [9].

The aim of this paper is to propose an automatic translation of programs’ sources into colored Petri nets for analysis purpose. Such an analysis can be performed to evaluate the behavior of these programs like we proposed in [10] for intrusion detection systems. To tackle the combinatory explosion of generated models as well as their analysis, we propose to consider separately various perspectives of the analyzed program. So, a property will be checked according to the corresponding perspective.

In addition to the theoretical description, this work has also been implemented as a tool named Evinrude. It basically takes a C program as input and produces a Petri net ready to be analyzed. The architecture of this tool and some experiments are presented in order to illustrate the approach.

This paper is structured as follows. Section 2 sketches a brief state of the art and presents our objectives. Section 3 presents the Evinrude tool. Section 4 explains how we extract the information produced by GCC. Exploitation of this information to produce and optimize Petri nets is detailed in sections 5 and 6. Finally, we apply our technique to an example in section 7.

2 Objectives

Several methodologies and techniques have been investigated for decades to produce correct software. The objective is to track bugs and imperfections, through program analysis.

2.1 Related work

Widely used in software engineering, static code analysis [11] is a technique that looks for patterns known to generate errors or bad behaviors during execution (such as bad pointer declaration, increment modification inside a loop, etc...) into either the source or the object code. Most of languages such as C/C++ [12] or Java [13, 14] are handled by common analyzers which are generally associated to compilers.

In addition to errors finding, static code analysis can also be used in association with formal methods that can mathematically prove properties about a given program. Hence, [15] defines formal software analysis and presents model checking [16] as a foundation technique for software engineering. Instead of considering source code only, this kind of analysis consists in systematically searching all possible behaviors of a system.

Model checking techniques are used by many software teams. Each one usually uses a dedicated modeling language. JavaPathFinder [17] or Bandera [18] translates Java source code directly into Promela language, the input language of
SPIN [7]. Feaver [19] produces Promela from C programs; here, model construction relies on user-specifications that describe pairs of C and Promela patterns. Another approach, SLAM [20], deduces predicate abstraction from a C program; these skeletons (they only contain boolean instructions) are then used as input to a dedicated model checker. VeriSoft [21] follows an approach base on a variation of previous model checking techniques. Indeed, it uses systematic testing at the implementation level. State-space exploration is performed by controlling and observing the execution of all visible operations in the concurrent processes of the system uses this kind of verification approach.

However, producing program’s abstractions for a model checker require skills in model building and also a deep understanding of the program: A bad interpretation of the source code leads to a less accurate model that has bad impact on the verification.

In addition to difficulties encountered during model construction, the size of the resulting model can also be a problem. For some programs (generally multi-threaded or parallel), we must consider the state space explosion problem. Common solution adopted by the community is to decrease the precision of the model, and thus, to reduce the precision of checked properties. Of course this trade-off is not satisfactory when dealing with system security.

2.2 Objectives of this work

To cope with these challenges, we aim at producing a framework dedicated to (i) program modeling and (ii) bugs and imperfections tracking through model analysis. It should be able to deal with large and/or multi-threaded programs.

For the modeling part, our objective is to build models to perform formal verifications. Model construction must be as automatic and precise as possible. Moreover, to handle large programs, we want to provide engineers with a flexible way to select sets of program’s specific behaviors they want to analyze.

For the analysis part, we focus on the reusability of existing analysis tools and methods. Our factory must build models that can be processed by an existing model checker (as it is done in most of the cited related works) and be used by usual tools dedicated to formal verification.

From all these requirements, we have selected Petri nets as the intermediate representation of the program. First, this formalism captures complex behaviors in a compact way. In particular, colored Petri nets are particularly adapted to handle parallel or multi-threaded behaviors (where similar patterns are executed concurrently). Second, the use of Petri nets (more specifically Symmetric nets1) let us benefit from the large collection of provided dedicated tools, like CPN-AMI [23] or GreatSPN [24].

1 Symmetric nets were formerly known as Well-Formed nets, a subclass of High-level Petri nets. The new name was chosen in the context of the ISO standardisation of Petri nets [22].
3 Tool Overview

We have designed a tool to achieve detection of bugs and imperfections in programs: *Evinrude*. It is made of several components as shown in figure 1. Each one deals with a dedicated transformation which is a part of the entire transformation process.

![Diagram of Evinrude tool's overview](image)

Fig. 1. *Evinrude* tool’s overview

We highlight three major steps in this process. First, the *parser module* analyzes the program’s sources and transforms them into an internal representation (a kind of rich abstract syntax tree) suitable for other modules (section 4). Then, each of these representations is interpreted by the *builder module* which produces a set of Petri nets (section 5). And finally, to reduce nets’ complexity, the *optimization module* applies reductions techniques (section 6).

The exhaustive analysis of an entire program is very difficult, if it is not impossible, since the number of instructions to be checked and properties to be verified is huge and not well bounded. Consequently, we define a *modular modeling* and a *modular analysis* dedicated to specific program’s behaviors. Given two different programs, the set of studied behaviors may not be the same: some of behaviors must be studied for both, others not. For example, some behaviors are only relevant for programs that provide networking or parallel features.

Thus, it’s up to engineers to decide what are the behaviors they want to study. To help them, we provide (packaged with the tool) a set of common behaviors that should be studied. Engineers can also define their own if it is necessary. This approach brings flexibility to our modeling and allows finer-grained analysis. Considering security related domain, common analyzed behaviors should be: system calls sequences (to avoid race conditions), synchronization mechanisms, array bounds (to avoid buffer-overflows), user-defined invariants (to guaranty security invariants) and i/o behaviors.

During the process, our framework considers program’s sources and instructions only if they are related to studied behaviors of the program. We call *perspective* each of these specific behaviors and *remarkable element* each related information extracted from the source code for this particular behavior. Each perspective has its own set of remarkable elements which could be words, structures or more complex patterns defined into the *perspective definition*.

---

2 Transformation rules are also defined in perspective definition (see section 5).
The final output of our production factory is a set of colored Petri nets: each one corresponding to a dedicated perspective. Once all nets have been generated, it is possible to merge some of them (or all of them) into a single Petri net to be processed by analysis tools.

4 Analyzing source files

Our first concern is to transform the source code into a representation that can be automatically analyzed by dedicated tools. Considering our modular objective, the parser module must select relevant information according to engineers specifications. Since we are not confident enough about compilation process, our factory uses an existing compiler framework to do the first part of this process while our component finishes the work.

4.1 Slicing the program using GCC

The parser module is a wrapper around the GNU Compiler Collection\(^3\) (GCC) in order to analyze source files. This choice gives us some independence from the programming language (C, C++, Java, etc.) thanks to the various front-ends available in this collection.

The very first operation, called slicing [25], is done by one of the many layers of GCC. Among GCC’s output, we exploit the link report, that defines relations between all source files, and the control flow graph (CFG) of the program, i.e., all paths that might be traversed through a program during its execution.

More precisely, GCC produces a file describing the program in terms of blocks linked together. These blocks are arranged according to the program’s control structures (functions, loops, conditionals) and are grouped into CFG functions. Listing 1.1 shows some parts of a CFG extracted by GCC. Functions and blocks are well visible. The corresponding C program is presented in section 7.

```
1. ;; Function philosopher1
2. # BLOCK 2
3. # PRED:ENTRY(fallthru)
4. printf(&"Philosopher 1..."
5. goto <bb 4> (<L1>);
6. # SUCC:4(fallthru)
7. # BLOCK 3
8. # PRED: 4 (true)
9. pthread_mutex_lock(&fork3);
10. pthread_mutex_lock(&fork1);
11. printf(&"Philosopher 1...
12. pthread_mutex_unlock(&fork3);
13. pthread_mutex_unlock(&fork1);
14. # SUCC:4(fallthru)
15. # BLOCK 4
16. # PRED:2(fallthru) 3(fallthru)
17. D.3892 = food_on_table();
18. if (f != 0) goto <L0>;
19. # BLOCK 2
20. # SUCC:3(true) 5(false)
21. # PRED:4(false)
22. # PRED:ENTRY(fallthru)
23. printf(&"Philosopher 1...
24. pthread_mutex_init(&food,0B);
25. # SUCC:EXIT
26. ;; Function main
27. # BLOCK 2
28. # PRED:ENTRY(fallthru)
29. pthread_mutex_init(&food,0B);
30. pthread_create
31. &phils [1];
32. D.3892 = &phils[1];
33. pthread_create
34. &phils[0B,philosopher1,0B);
35. D.3880 = &phils[1];
```

Listing 1.1. CFG produced by GCC

\(^3\) The parser module uses the fdump option of GCC available since version 4.0.
After the slicing operation all control structures have been rewritten and included in the CFG representation. Consequently, we do not deal anymore with control structures like `for`, `while`, `continue`, `break`... but only with block sequences and function links. Even “evil sequences” are simplified by this operation.

### 4.2 Building a suitable representation

The builder and the optimization modules cannot directly use the CFG representation (even if it is much simpler than the source code itself). Thus, the parser module has to transform all gathered information into an internal representation suitable for other modules (see figure 2).

Objects `Assignment` and `Call` are more complex in the tool than presented here. However, considering the example and the selected perspectives, this representation is detailed enough to understand the process.

While building this representation the parser module extracts a symbols table and a set of statistics about the program and all its dependencies. It also gives advices on what perspectives should be requested for building and analysis. This list only includes basic perspectives, and engineers can add theirs if necessary. Table 1 presents these results for the studied example program.

**Table 1.** Information returned by the parser module

<table>
<thead>
<tr>
<th>Application's name:</th>
<th><code>dining_philosopher</code> (1 file)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of processed CFGs:</td>
<td>1 (320 instructions)</td>
</tr>
<tr>
<td>Number of processed functions:</td>
<td>8 (5 connected - 1 main function)</td>
</tr>
<tr>
<td>Recommended perspective selection:</td>
<td>- structural perspective (<code>struct</code>)</td>
</tr>
<tr>
<td></td>
<td>- system calls perspective (<code>syscall</code>)</td>
</tr>
<tr>
<td></td>
<td>- pthread perspective (<code>thread</code>)</td>
</tr>
</tbody>
</table>

The representation can be saved into a XML file to be processed or reprocessed later without having to parse again the entire application source code.
5 Generating Petri Nets

As soon as the internal representation has been produced by the parser module, the builder module produces a model for each selected perspective. The struct perspective is always processed (and processed first) by the builder module since it deals with the program structure. The result is the skeleton of the final model.

Models, generated using other perspectives, are linked to this skeleton. The structural model and all other perspective’s models are then flattened to produce the final Petri net (see figure 3).

In addition to remarkable elements, a perspective definition comes with a set of transformation rules that associate remarkable elements to a set of actions to be applied on the Petri net model (see section 3). The builder module uses these transformation rules to produce all models.

A rule is defined by the following elements:
– its identifier and title
– if necessary, preconditions that must be satisfied prior to application
– the transformation algorithm

Instead of directly manipulating Petri nets, the builder module uses an internal representation using objects (see figure 4). This representation is dedicated to manipulation and optimization of the model.

This representation brings hierarchy to Petri nets. Indeed, a model (or a submodel) can contain one or more submodels connected by means of places. A place can also refer to another one to make links between submodels.
5.1 The structural perspective

The structural (struct) perspective contains seven rules. The first two rules (Struct1 and Struct2) are dedicated to blocks management.

**Struct1: CFG blocks.**
We associate a place \( F_X \) to each block \( X \) of a function where \( F \) is the function’s identifier calculated by the parser module.

\[
\begin{array}{c}
\text{struct}\_4\_4\_3 \\
4\_3 \\
\end{array}
\]

**Struct2: CFG functions.**
We create two places for each analyzed function \( F \) in the CFG. They are labeled \( F_{\text{entry}} \) and \( F_{\text{exit}} \) where \( F \) is an ID attributed by the parser module.

\[
\begin{array}{c}
4\_\text{entry} \\
4\_\text{exit} \\
\end{array}
\]

**Notes:** Illustration of rules are based on lines 1 to 20 of the CFG in listing 1.1⁴. From now, we call function \( F \) (resp. block \( X \)), the function (resp. the block) whose identifier, attributed by the parser module, is \( F \) (resp. \( X \)).

The next two rules (Struct3.1 and Struct3.2) deal with links between blocks. The first rule browses blocks by following all successors paths. The second one deals with predecessors paths.

**Struct3.1: Successor links.**
Given a block \( X \) of a function \( F \). For each block’s successor \( Y \), we create a transition labelled \( \text{struct}\_F\_X\_Y \). Finally we link the place associated to the block \( X \) to the new transition, and the new transition to the place designated by \( Y \).

\[
\begin{array}{c}
\text{struct}\_4\_4\_3 \\
4\_4\_5 \\
\end{array}
\]

**Struct3.2: Predecessor links.**
Given a block \( X \) of a function \( F \). For each block’s predecessor \( Y \), we create a transition (if it does not already exist) labelled \( \text{struct}\_F\_Y\_X \). Finally we link the place associated to \( Y \) to the new transition, and the new transition to the place designated by \( X \).

\[
\begin{array}{c}
\text{struct}\_4\_\text{entry}\_0 \\
4\_2 \\
\end{array}
\]

Rule Struct4 creates links between functions. A link between two functions exists only if both have an internal representation available (i.e. their source code is in the program and not in a library). Otherwise, the call is considered as an external call, and might be processed later by another perspective. The Petri net example for this rule is built from line 17 of listing 1.1⁵.

**Hierarchy:** This transformation rule introduces a hierarchy in the model according to precise guidelines:

- Each subnet is identified by an unique identifier derived from the instruction counter of the CFG. This identifier is often written as \( I \) in rules definition.
- Each subnet has a set of incoming places and outgoing places. These places are used to merge the subnet into the main net at the end of the build stage.
- A subnet can contain virtual places that refer to existing places inside the main net. Virtual places and their references are merged at the end of the build stage too.

---

⁴ The parser module has attributed id 4 to philosopher1 function

⁵ The parser module has attributed id 7 to the function food_on_table()
Struct4: Links between functions.

Given a block $X$ of a function $F$ calling a function $G$. We define a subnet associated to the structural place $F \_ \_X$ defined as follows: we create two places labeled $F \_ \_X \_ \_G \_ \_call$ and $F \_ \_X \_ \_G \_ \_return$. The first one is an incoming place. It is linked to a transition $call \_F \_ \_X \_ \_G$ which is linked to the virtual place $G \_ \_entry$. For the return path, the virtual place $G \_ \_exit$ is linked to a transition $return \_F \_ \_X \_ \_G$ which is linked to the place $F \_ \_X \_ \_G \_ \_return$. This last place is an outgoing place.

Rule Struct4 does not fully control return path constructions. If two functions $A$ (id=1) and $B$ (id=2) make a call to function $C$ (id=0), the net produced by Struct4 (figure 5(a)) allows bad return path. In the example, the sequence: call_1_X_0, return_2_X_0 is a mistake but is allowed in the resulted model.

(a) Bad return path
(b) Controlled return path

Fig. 5. How to control the return path

Rules Struct5 and Struct6 address this problem. The first one controls the function’s return path. The second one handles recursive constructions by controlling nested function calls and returns.

Struct5: Return management.

Each function call from $F$ is associated with a place $path \_F \_C$ where $C$ is a unique call identifier calculated by the parser module. We create a link from $call \_F \_X \_ \_G$ to this new place and from this new place to the $return \_F \_X \_ \_G$ transition (see Struct4).

Struct6: Handle nested calls.

For each $return \_F \_X \_ \_G$ transition, we create a link from place $path \_G \_* \_ \_X$ to the return transition. Instead of a simple arc, this link is an inhibitor arc. Thus the transition is fireable only if all places $path \_G \_* \_ \_X$ are empty.

Initial marking: Finally, the initial marking is set up. A single non-colored token corresponding to state of the program is put into place $X \_ \_entry$ where $X$ is the identifier of the main function.
5.2 The system call perspective

System calls are remarkable elements of the syscall perspective. For each system call, a dedicated subnet is created and associated with a place of the structural model (rule Syscall0).

<table>
<thead>
<tr>
<th>Syscall0: System call inclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given a system call $S$ in a block $X$ of a function $F$. We associate a place $F _X _I _pre$, a transition $sys _F _X _I _S$ and a place $F _X _I _post$ (where $I$ is the unique CFG’s instruction identifier) with the structural place $F _X _F _X _pre$ marked as an incoming place, $F _X _I _post$ marked an outgoing place. We also create arcs along the path $F _X _I _pre \rightarrow sys _F _X _I _S \rightarrow F _X _I _post$.</td>
</tr>
</tbody>
</table>

5.3 The thread perspective

Management of threads inside a C program usually involves three primitives: pthread_create, pthread_exit and pthread_join. The thread-perspective is designed to handle this kind of behaviors. Other primitives like pthread_kill or pthread_self, which are also defined in POSIX library, are not handle by this perspective since they do not directly modify the behavior of a thread. Moreover, since it exists three different kinds of thread mutexes, we choose to only analyze the behavior of the most portable one (that is used by default when dealing with the library): PTHREAD_MUTEX_NORMAL

Rule Thread0 handles the creation of dedicated subnets, associated to a structural place. As other perspectives, these subnets are merged at the end of the build phase (see section 5.4).

<table>
<thead>
<tr>
<th>Thread0: Thread primitives inclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given a thread management primitive $S$ in a block $X$ of a function $F$. We associate a place $F _X _I _pre$, a transition $thread _F _X _I _S$ and a place $F _X _I _post$ (where $I$ is the unique CFG’s instruction identifier) with the structural place $F _X _F _X _pre$ marked as an incoming place and $F _X _I _post$ place is marked as an outgoing place. We also create arcs along the path $F _X _I _pre \rightarrow thread _F _X _I _S \rightarrow F _X _I _post$.</td>
</tr>
</tbody>
</table>

Rule Thread1 handles thread creation. A new thread is modeled as a new token in the Petri net. Each token represents the state of a particular thread as the program counter does in the system.
Thread1: Thread creation

The transition `thread_F_X_I_pthread_create` (previously created by the rule Thread0) produces a token into the `G_entry` place, where `G` is the identifier of function which be executed by the thread (third parameter of the call). We create a virtual place that refers the `G_entry` place and an arc between `thread_F_X_I_pthread_create` transition and this virtual place.

Rules Thread2 and Thread3 deal with thread termination. In a system, when a thread dies, a part of its information is kept until another thread joined it. After that, dead thread’s information is totally removed from the system.

Thus, when a thread ends, its corresponding token is put into the `exit` place of the `main` function, waiting to be retrieved (joined) by another thread (see rule Thread2). The `join` operation (which is a system blocking operation) is modeled as trying to get a thread token from the `exit` place of the `main` function (see rule Thread3).

Thread2: Thread ending

The transition `thread_F_X_I_pthread_exit` puts the thread token into the exit place of main function. So, we create a virtual place that refers `G_exit` where `G` is the identifier of `main` function. Finally, we create an arc from `thread_F_X_I_pthread_exit` transition to this virtual place.

Thread3: Thread joining

The transition `thread_F_X_I_pthread_join` consumes a token from the exit place of the main function. So, we create a virtual place that refers `G_exit` where `G` is the identifier of `main` function. Finally, we create an arc from this virtual place to `thread_F_X_I_pthread_join` transition.

Synchronization: Thread synchronization is handled by means of five primitives dedicated to lock management.

- `pthread_mutex_init()` and `pthread_mutex_destroy()`
- `pthread_mutex_lock()` and `pthread_mutex_trylock()`
- `pthread_mutex_unlock()`

Rules ThreadSynchro1 and ThreadSynchro2 focus on mutex creation and destruction. Mutexes are represented by a colored place typed by:

\[ C_{mutex} = \{lock, free\} \]

Status of a mutex is represented as follow:

- the mutex is locked when its corresponding place contains token `lock`,
- the mutex is unlocked when its corresponding place contains token `free`,
- the mutex has not been initialized when its corresponding place is empty.
**ThreadSynchro1: Create a lock**

We associate an empty place mutex \( K \) to each created lock, where \( K \) is a unique identifier computed by the parser module. We create an arc between the transition \( \text{thread}_{F_X,I} \cdot \text{pthread_mutex_init} \) and the mutex \( K \) place. The arc’s valuation is set to \( \text{free} \) meaning that the mutex is free.

**ThreadSynchro2: Destroy a lock**

Transition \( \text{thread}_{F_X,I} \cdot \text{pthread_mutex_destroy} \) consumes the token from the place mutex \( K \) whatever its value is. We create an arc between these two objects. Its valuation is set to 'e. Obviously, any further access will generate a blocking state. This particular state can be detected at verification time.

Locking and unlocking primitives are handled by the next two rules. We distinguish two behaviors when a lock is set. The first one (lock primitive) implies a blocking state when the lock is already taken. The second one (trylock primitive) is a non-blocking operation. Thus we design two versions of the rule **ThreadSynchro3** to handle these two behaviors.

**ThreadSynchro3.1: Take a lock**

The transition \( \text{thread}_{F_X,I} \cdot \text{pthread_mutex_lock} \) consumes the colored token from the place mutex \( K \). If the value of the token is \( \text{free} \) the transition can be fired. Otherwise, the transition is disabled. The transition’s guard is set to \( [v = \text{free}] \). If fired, the transition puts a lock token into the mutex \( K \) place.

**ThreadSynchro3.2: Try to lock**

Transition \( \text{thread}_{F_X,I} \cdot \text{pthread_mutex_trylock} \) consumes the token from the place mutex \( K \). Whatever the value of the token is, the transition is fired. After the firing, a lock token is put into the mutex \( K \) place.

Releasing a lock consists in putting a free token into the mutex place.

**ThreadSynchro4: Release a lock**

The transition \( \text{thread}_{F_X,I} \cdot \text{pthread_mutex_unlock} \) consumes the token of place mutex \( K \). Whatever its value is, the transition is fired and a free colored token is put into the place mutex \( K \).

5.4 Merging perspectives

At the end of the building phase, the resulting Petri net is made of:

- a skeleton, thanks to structural perspective;
- several subnets sticked to some structural places.
Since we do not perform analysis on our hierarchical model yet, we use an algorithm (see algorithm 1) to flatten the model. This algorithm replaces structural places by a composition of all contained subnets. Once the model is flat, it can be reduced and then analyzed.

**Algorithm 1:** Flatten the produced model

```plaintext
foreach p ← place of main net do
    if p contains at least one subnet then
        outgoings ← p
        outputs ← getOutputTransitions(p)
        foreach s ← subnet inside p do
            {copy all elements from the subnet to the main net}
            copyAll(s, main)
            {create links between two sets}
            createLinks(outgoings, getIncomingsPlaces(s))
            outgoings ← getOutgoingsPlaces(s)
        end
        createLinks(outgoings, outputs)
    end
{merge virtual places with their references}
foreach v ← getVirtualPlaces(main) do
    ref ← findPlaceByName(main, getReferenceName(v))
    mergeTwoPlaces(v, ref)
end
```

6 Reducing Petri Nets

Because of all perspective’s production rules, our factory produces detailed models that are generally quite large (see table 6). But, while places and transitions that come from perspectives selected by engineers are useful for analysis, objects from the structural model are not relevant anymore; especially because these elements do not correspond to remarkable elements we want to observe.

Thus, we remove these useless parts in the model to reduce its size. To do so, we apply a set of reductions rules on the produced model. The two first rules are slightly adapted from the ones of Haddad [26] to fit our strategy. The third one detects a typical configuration that frequently happens in our models.

**Pre-agglomeration of transitions** aims at reducing sequences of transitions. When a place $p$ is accessed by the firing of a transition $t$ and left by the firing of any output transition of $p$, we delete the place $p$ and the transition $t$ and make arcs between input places of $t$ and output transitions of $p$. This reduction is valid only with transitions labeled as `struct X,Y,Z`.

**Post-agglomeration of transitions** is dedicated to conditional structures. When all branches of a transition $t$ join up with the end of a block (which is
linked to another block), we directly link all branches to the next block. The intermediate place is deleted.

**Diamonds reduction** concerns control structures like `switch`. When no remarkable elements is located in `case` blocks, a place $p_1$ is linked to several transitions $t_i$ which are all linked to a place $p_2$. We call this configuration a **diamond**. Since all paths are equivalent in our model, we merge all transitions into a single one. We then obtain the sequence: $p_1$ linked to $t$ linked to $p_2$.

### 7 Application to an example

We use a simple C program implementing the well known philosopher problem [27] to illustrate our approach. This code is a variation from the one used by Sun Microsystems to benchmark their Thread analyzer [28].

#### 7.1 The program

The program is presented in listing 1.2. The only change we made to original program is a duplication of the philosopher code (presence of three `philosophers` functions) to avoid data-flow analysis that is not yet processed by Evinrude.

```c
#define FOOD 50

pthread_mutex_t foodlock;

pthread_mutex_t fork1, fork2, fork3;

pthread_t p[3];

void * philosopher1 ();
void * philosopher2 ();
void * philosopher3 ();

int food_on_table ();

int main (int argc, char **argv) {
  int i;

  pthread_mutex_init(&foodlock, NULL);

  pthread_mutex_init(&fork1, NULL);
  pthread_mutex_init(&fork2, NULL);
  pthread_mutex_init(&fork3, NULL);

  pthread_create(&p[0], NULL, philosopher1, NULL);
  pthread_create(&p[1], NULL, philosopher2, NULL);
  pthread_create(&p[2], NULL, philosopher3, NULL);

  pthread_join(phils[0], NULL);
  pthread_join(phils[1], NULL);
  pthread_join(phils[2], NULL);
}

void * philosopher1 () {
  int f;

  printf("Philo␣1␣sitting ␣down␣to␣dinner.
");
  while((f = food_on_table())) {
    pthread_mutex_lock(&fork3);
    pthread_mutex_lock(&fork1);
    printf("Philo␣1␣eating.
");
    pthread_mutex_unlock(&fork3);
    pthread_mutex_unlock(&fork1);
  }
  printf("Philo␣1␣is␣done␣eating.
");
  pthread_exit(NULL);
}

void * philosopher2 () {
  int f;

  printf("Philo␣2␣sitting ␣down␣to␣dinner.
");
  while((f = food_on_table())) {
    pthread_mutex_lock(&fork1);
    pthread_mutex_lock(&fork2);
    printf("Philo␣2␣eating.
");
    pthread_mutex_unlock(&fork1);
    pthread_mutex_unlock(&fork2);
  }
  printf("Philo␣2␣is␣done␣eating.
");
  pthread_exit(NULL);
}

void * philosopher3 () {
  int f;

  printf("Philo␣3␣sitting ␣down␣to␣dinner.
");
  while((f = food_on_table())) {
    pthread_mutex_lock(&fork2);
    pthread_mutex_lock(&fork3);
    printf("Philo␣3␣eating.
");
    pthread_mutex_unlock(&fork2);
    pthread_mutex_unlock(&fork3);
  }
  printf("Philo␣3␣is␣done␣eating.
");
  pthread_exit(NULL);
}

int food_on_table () {
  static int food = FOOD;

  pthread_mutex_lock(&foodlock);
  if (food > 0) { food --; }
  myfood = food;
  pthread_mutex_unlock(&foodlock);
  return myfood;
}
```

**Listing 1.2.** A philosopher program

Basically, the main program starts several threads executing a `philosopherX` function implementing one philosopher’s behavior.
First, each philosopher looks for food on the table (the `food_on_table` function reads a shared resource protected by a mutex). If there is food, the normal process takes place and the philosopher tries to take two forks (which are distinct mutexes). If not, the philosopher leaves the table (thread exits). At the end, the main thread catches (join) all ended threads and terminates.

7.2 The generated Petri net

The dining philosopher program has been processed by Evinrude. Perspectives `structural`, `syscall` and `threads` have been selected. The resulted Petri net is displayed in figure 6. It has 36 places, 33 transitions and 110 arcs. The computation takes less than 10 seconds including GCC analysis.

Fig. 6. The Petri net produced from the philosopher example
We have drawn areas in the Petri net to outline major components they represent according to the original program’s function:

- thick plain places represent mutexes,
- zone A corresponds to the \textit{philosopher1} function,
- zone B corresponds to the \textit{philosopher2} function,
- zone D corresponds to the \textit{philosopher3} function,
- zone C corresponds to the \textit{food on table} function,
- other places correspond to the \textit{main} function.

Table 2 shows the reduction factor of the model during the optimization phase.

\begin{table}
\begin{center}
\begin{tabular}{|c|c|c|c|}
\hline
& \textbf{Before optimization} & \textbf{After optimization} & \textbf{Reduction factor} \\
\hline
Places & 99 & 36 & 63\% \\
Transitions & 91 & 33 & 63\% \\
Arcs & 226 & 110 & 51\% \\
\hline
\end{tabular}
\end{center}
\end{table}

7.3 Some analysis of the model

Some bad constructions such as structural infinite loops or structural dead code can be automatically detected on the model. Infinite loops corresponds to a cycle without any exit condition in the Petri net model. Structural dead code corresponds to a subnet that is not connected to the main one.

\begin{table}
\begin{center}
\begin{tabular}{|c|c|c|c|c|}
\hline
\textbf{Terminal state \#1} & \textbf{Marked} & \textbf{Terminal state \#2} & \textbf{Marked} \\
\hline
Marked place name & \textit{Marking} & Marked place name & \textit{Marking} \\
State\_3\_exit & \textbullet & State\_3\_2\_121\_post & \textbullet \\
State\_mutex\_fork2 & \textit{free} & State\_mutex\_fork2 & \textit{lock} \\
State\_mutex\_fork1 & \textit{free} & State\_4\_3\_151\_post & \textbullet \\
State\_mutex\_fork1 & \textit{free} & State\_mutex\_fork1 & \textit{lock} \\
State\_mutex\_foodlock & \textit{free} & State\_mutex\_fork3 & \textbullet \\
State\_mutex\_foodlock & \textit{free} & State\_5\_3\_193\_post & \textbullet \\
State\_6\_3\_235\_post & \textbullet & State\_mutex\_foodlock & \textbullet \\
\hline
\end{tabular}
\end{center}
\end{table}

For fine grained analysis, we use the CPN-AMI [23] Petri net tools. Let us generate the state space and check for any terminal state. The state space produced by Prod [29] in CPN-AMI has 845 nodes and 2413 arcs. Two terminal states are detected. They are presented in table 3.
Terminal state #1 corresponds to a normal end of the program since the exit place of the main function (3.exit) has only one token (there is no marking in the threads subnets) and all shared resources are free.

Terminal state #2 corresponds to a deadlock. The program’s state is the following (we provide instruction identifiers in the CFG when relevant):

– all forks are handled by philosophers,
– philosopher 1 is waiting on instruction #151,
– philosopher 2 is waiting on instruction #193,
– philosopher 3 is waiting on instruction #235,
– main thread is waiting on instruction #121.

Evinrude is able to recognize an instruction identifier and to provide engineers with its translation into C code. This code is extracted from the detailed CFG produced by GCC. Thus, some syntax variations can happen. In the example, table 4 shows associations returned by the tool:

Table 4. Relation between instruction identifier and C code

<table>
<thead>
<tr>
<th>State ID</th>
<th>Associated C code</th>
</tr>
</thead>
<tbody>
<tr>
<td>121</td>
<td>pthread_join (phils[0], OB);</td>
</tr>
<tr>
<td>151</td>
<td>pthread_mutex_lock (&amp;fork1);</td>
</tr>
<tr>
<td>193</td>
<td>pthread_mutex_lock (&amp;fork2);</td>
</tr>
<tr>
<td>235</td>
<td>pthread_mutex_lock (&amp;fork3);</td>
</tr>
</tbody>
</table>

The tool also highlights a sequence of 22 transition firings that leads to this deadlock. From such a path, it is possible, as done before, to locate, outline and animate the corresponding instructions in the C program by using the information stored in the detailed CFG. Table 5 produces this trace in an comprehensive way for an engineer.

7.4 Transforming larger programs

Since our global approach is dedicated to intrusion detection systems, our benchmarks are more system-oriented than the philosopher problem and difficult to present in a regular paper. Indeed, attackers often exploit flaws in programs to get control of a computer, stole important data, etc. Generating a model of such programs can lead to detection of flaws or possible entry points for attackers and to prevent these kinds of attacks.

Evinrude is able to deal with large and real programs. Here are some examples of programs we have processed considering only the struct, syscall, processes and thread perspectives.

gzip (v1.2.4) is a compression utility included in most of Unix systems. It has been adopted by the GNU projects. It also often uses by FTP servers.
Table 5. A sequence of instructions leading to the deadlock

<table>
<thead>
<tr>
<th>Main</th>
<th>Philo 1</th>
<th>Philo 2</th>
<th>Philo 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_mutex_init (&amp;foodlock, OB); (#112)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pthread_mutex_init (&amp;fork1, OB); (#113)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pthread_mutex_init (&amp;fork2, OB); (#114)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pthread_mutex_init (&amp;fork3, OB); (#115)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pthread_create (&amp;phils, OB, philosopher1, OB); (#116)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pthread_create (D.3880, OB, philosopher2, OB); (#118)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pthread_mutex_init (&amp;foodlock, OB); (#202)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pthread_mutex_lock (&amp;foodlock); (#274) (v=free)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pthread_mutex_unlock (&amp;foodlock); (#291) (v=lock)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>return of food on table function; (#202)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pthread_mutex_lock (&amp;fork1); (#192) (v=free)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pthread_mutex_init (D.3881, OB, philosopher3, OB); (#120)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pthread_mutex_init (&amp;foodlock, OB); (#202)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pthread_mutex_lock (&amp;foodlock); (#274) (v=free)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pthread_mutex_unlock (&amp;foodlock); (#291) (v=lock)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>return of food on table function; (#202)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pthread_mutex_lock (&amp;fork2); (#150) (v=free)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pthread_mutex_init (D.3891, OB, philosopher3, OB); (#120)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pthread_mutex_init (&amp;foodlock, OB); (#202)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pthread_mutex_lock (&amp;foodlock); (#274) (v=free)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pthread_mutex_unlock (&amp;foodlock); (#291) (v=lock)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>return of food on table function; (#202)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pthread_mutex_lock (&amp;fork3); (#150) (v=free)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

wu-ftpd (v2.6.2) is a FTP server software for Unix systems. Up until early 2000s, it was the most common FTP server software in use.

lighttpd (v1.4.19) is light web-server that has been designed for high-performance environments. It has also a low memory footprint.

Data about Petri nets generated from these three programs is provided in table 6.

Table 6. Modeling results for some UNIX programs

<table>
<thead>
<tr>
<th>Program’s size (lines)</th>
<th>gzip</th>
<th>wu-ftpd</th>
<th>lighttpd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program’s size (lines)</td>
<td>7 788</td>
<td>18 405</td>
<td>52 336</td>
</tr>
<tr>
<td>Model (^6)</td>
<td>842/1 119/2 406</td>
<td>1 312/5 331/1 1754</td>
<td>3 403/4 264/8 399</td>
</tr>
<tr>
<td>Optimized model (^6)</td>
<td>149/165/498</td>
<td>829/963/3 018</td>
<td>673/761/2 392</td>
</tr>
</tbody>
</table>

8 Conclusion

In this paper, we have presented Evinrude, a tool that translates a C program into colored Petri nets for analysis purpose. Such an analysis is operated in the context of Intrusion Detection Systems (IDS) where it is of interest to check programs with regards to “dangerous” behaviors. This technique, called off-line monitoring, is similar to performing model checking on programs.

\(^6\) In terms of places/transitions/arcs
We use GCC as a front-end to perform program slicing. We exploit information from the Control Flow Graph (CFG) to produce our Petri nets. So, if experimentation in the paper is done on C programs, our technique should be applicable to any language processed by GCC without majors changes.

To reduce the size of the resulting Petri nets, we consider separate perspectives on a program. A perspective groups remarkable elements to be observed in the target Petri net model. Perspectives can be operated separately or chained, according to what has to be observed.

Our approach relies on the perspective notion. A perspective is a way to aggregate transformation techniques dedicated to a purpose (i.e. system calls, synchronization, etc). Perspectives can be elaborated and adapted according to a new purpose and provides flexibility for program analysis. Perspectives may also be used separately (to focus on one aspect to be analyzed). Finally, they can be composed when several aspects of a program must be analyzed simultaneously (because they interact).

So, Evinrude’s transformation process relies on rules associated to a perspective. Our Petri net generator applies the rules associated to the selected perspective. Once the Petri net is generated, we apply an optimization phase that mainly relies on Haddad’s reductions [26].

The way we can create, select and compose perspectives in Evinrude allows one to control the complexity of specifications produced from source code.

References

28. Microsystem, S.: Sun studio express - thread analyzer readme
Quickly prototyping
Petri nets tools with SNAKES

Franck Pommereau
LACL, Université Paris Est
61 avenue du Général de Gaulle
94010 Créteil, France
pommereau@univ-paris12.fr

Abstract. This paper presents the toolkit snakes that is aimed at providing a flexible solution to the problem of quickly prototyping Petri nets tools. In particular, snakes is expected to have as few built-in limitations as possible with respect to the particular variant of Petri net to be used. The goal is to make snakes suitable for any kind of Petri net model, including new ones for which there exists no available tool. For this purpose, snakes is designed as a very general Petri net core library enriched with a set of extension modules to provide specialised features. On the one hand, the core library is versatile in that it defines a general Petri net structure where all the computational aspects are delegated to an interpreted programming language. On the other hand, extension modules provide with enough flexibility to allow to redefine easily any part of the base Petri net model. In particular, snakes comes with a handful of extension modules in order to handle models of the Petri Box Calculus and M-nets family.

snakes is released under the GNU LGPL, it can be downloaded, together with API documentation and a tutorial, at ⟨http://lacl.univ-paris12.fr/pommereau/soft/snakes⟩.

Key words: Petri nets, quick prototyping.

1 Introduction

There exists a wide range of Petri net tools [24], most of them (if not all) being targeted to a particular variant of Petri nets or a few ones. When a new interesting variant is defined, it is often necessary to develop a software to support it, and this tool has to be updated as the model and associated techniques evolve. When research is targeted on a defined usage, as model-checking for instance, the formalism is often fixed and this situation causes no problem. The tool is simply improving over time. But when research is centred on evolutions of the model itself, a tool often has a very short lifetime. It becomes then very hard for developers to keep the pace with theory and often it does not worth the effort as the tool will not be used anymore when the next variant of the model will be defined.

snakes is an attempt to solve this problem by providing a general and flexible Petri net library allowing for quick prototyping and development of ad-hoc and test tools. The requirements for such a toolkit may be as follows:

1. Built-in Petri net model. This is the most obvious need.
2. General and flexible. The toolkit should be able to cope with a large variety of Petri net models. Moreover, it should be easy to extend it with new variants of Petri nets.
3. Easy to use and portable. The goal being to be able to quickly implement new ideas, it should not be intimidating to start programming, so the toolkit must be easy to understand. It should be also easy to install it anywhere, as well as resulting programs.1
4. Intended for prototyping. This requirement alleviates the question of performances and solves a contradiction that would arise otherwise: a flexible and general tool with dynamic reconfiguration of its features would be hard to make fast.
5. Interoperable with other tools. One tool cannot solve all the problems, so it is necessary that a toolkit can collaborate with other tools, in particular through importing/exporting PNML.

It is a well known pattern for programs that need to have few built-in limitations to define a general framework with the basic required features and to provide then scripting capabilities allowing

---

1 This is actually a general requirement as if a software is complicated and works on a very specific platform, it is likely that only few people will use it.
to extend the tool or redefine parts of it. This is the case for instance for text editor emacs or typesetting system \TeX/\LaTeX, both being often perceived as tools with unlimited features. snakes follows this pattern by defining a simple but very general Petri net structure: places, transitions, arcs, tokens. This is indeed what all Petri nets have in common. At this level of generality however, not many features are available. Then, Python [35] has been chosen as extension language. Python is a mature, well established, interpreted language, that has all the required characteristics to meet the needs expressed above. According to its web site:

*Python is a dynamic object-oriented programming language that can be used for many kinds of software development. It offers strong support for integration with other languages and tools, comes with extensive standard libraries, and can be learned in a few days. Many Python programmers report substantial productivity gains and feel the language encourages the development of higher quality, more maintainable code.*

It may be added that Python is free software and runs on a very wide range of platforms. In order to provide with (hopefully) unlimited scripting capabilities, snakes delegates all the computational aspects of Petri nets to Python. In particular, a token is an arbitrary Python object, transitions execution can be guarded by arbitrary Python Boolean expressions, and so on. As a result, a Petri net in snakes is mainly a skeleton with very general behavioural rules (consume and produce tokens in places through the execution of transitions) and with the full power of a programming language at any point where a computation is required. snakes itself is programmed in Python and uses the capability of the language to dynamically evaluate arbitrary statements. Using the same programming language for snakes and its extension language is a major advantage for the generality: Petri nets in snakes can use snakes as a library and work on Petri nets. For instance, as a token in snakes may be any Python object, it could be an instance of the Petri net class of snakes.

Snakes is more particularly targeted to the family of the Petri Box Calculus (pbc) [5,6] and M-nets [7,26] for which Petri nets may be composed as terms in a process algebra. Many variants of the base model exist and new ones are still under development, each being focused on the study of a particular feature (e.g., time, preemption, threads, exceptions, . . .). In order to provide support for these models without specialising the general framework presented above, snakes comes with various extension modules (also called plugins) to address the various aspects of these models. This plugin system is a simple and convenient way to extend and specialise the core library. Many short-life tools or prototypes can be developed as plugins; this was made for instance during a work about verification of Petri nets equipped with unbounded integer variables [34]. This particular case is related in section 3.1 below where perceived benefits of prototyping with snakes in parallel with writing the paper are explained.

All but one of the requirements listed above have been discussed so far. The ease to use has been tested with four students in computer science, at the end of their first year of master degree. All four were average students having no prior knowledge of Python and only basic notions about Petri nets. They were given a one hour presentation of Python and snakes and provided with the Python tutorial. Each of them had to implement an algorithm described in a paper: McMillan’s unfolding [27], Finkel’s version of Karp & Miller’s coverability graph [22], M-nets unfolding to p/t-nets [7] and Petri net semantics of mins [31]. All the four students could provide a working program after a few weeks of work, none of them did request for help except to understand the papers, and the quality of their programs distributed evenly from acceptable to excellent. Snakes has been used also in a teaching context, in order to introduce practical works in a theoretical course about Petri nets. The requirement was to model and verify a simple system, e.g., an elevator or a simplified atm. Snakes has been used to build basic Petri nets, compose them and search their reachable markings for faulty states. In this experience also, students have had no problem using snakes.

### 1.1 Use cases and performance issues

Snakes has been used to implement various Petri nets semantics:

- the Causal Time Calculus [32] is a pbc-like process algebra with a Petri net semantics that has been implemented on the top of snakes in less than 200 straightforward lines of code;
- boon [10] is an object-oriented programming notation with a Petri net semantics that has been implemented using snakes called from a Java program;
Quickly prototyping Petri nets tools with SNAKES

– a Petri net semantics of MINS interconnection networks has been implemented and used for simulation [31];
– a Petri net semantics of the Security Protocol Language (SPL) [9] has required also about 200 lines of code;
– a compiler for the Asynchronous Box Calculus with Data (abcd) [11] has been implemented in snakes and is provided in the distribution (see section 5).

In all these cases, performances were not an issue as the goal was to use the ability of SNAKES to perform various compositions on Petri nets. But it was not used to execute nets produced (except for MINS where execution time was not critical). This kind of application is actually the main intended usage for SNAKES that was started in order to support models from the PBC and M-nets family. It is often the case when working with this family that the main issue is to build a Petri net that is later verified using a specialised tool. This holds for instance with the semantics of SPL that is computed using SNAKES, yielding a Petri net that is translated to Helena [19] formalism for fast verification.

One other use case of SNAKES is developed more in details in section 3.1 and consisted in implementing a state space construction. In this case also, performances were not an issue: most of computation involved when building state space was delegated to a library implemented in C. Computation performed by Python was small enough to allow us to run our examples and validate our algorithms.

More generally, SNAKES should be efficient enough when used to build Petri net semantics of systems, or to perform interactive simulation. For more computationally intensive applications, several generic solutions may speed up SNAKES:

– Psyco [37] is a kind of just-in-time compiler for Python, it provides a typical speedup of 4;
– Shed Skin [17] allows to compile restricted Python to C++, which should allow to translate parts of SNAKES. Typical speedup is about 35;
– PyPy [12] is able to execute full Python with a just-in-time code generator, or to compile it to C. In the later case, typical speedup is 65.

These are easy to use but limited solutions. More generally, the approach in order to get a real application from a prototype using SNAKES would be either:

– delegate heavy computation to an external fast library (like presented in section 3.1), or,
– profile the prototype and implement a fast optimised version of critical parts as an external library, which allows to fallback to previous case.

It is very unlikely that a whole prototype has to be reimplemented because there will always exists parts of it that do not need a fast implementation. For instance, this is usually the case for import/export to PNML or other formats that is I/O bound rather than CPU bound.

1.2 Related tools

Several existing tools may be related in particular to SNAKES. The first one is the Petri net kernel (PNK) [36] that shares with SNAKES the aim to provide a general framework for building Petri nets applications. The PNK provides a graphical user interface for editing and simulating Petri nets, its main aim being to provide basis to real applications rather than to allow for quick prototyping. With respect to SNAKES, the basic model of the PNK is a less general model of coloured Petri nets; however, this may be extended by writing Java code. Another difference is that the PNK does not provide any of the operations in order to manage models from the PBC and M-nets family, which is of course not its aim. The development of the PNK does not appear to be active anymore, the last release being dated of March 2002. The PNK is free software distributed under terms of the GNU GPL, which forces tools that use the PNK to be released under the same licence. SNAKES uses GNU LGPL, which is less restrictive and allows to produce non-free software that uses SNAKES, but forces to release under GNU LGPL any change made to SNAKES.

Another tool with which SNAKES shares some goals is the Programming Environment based on Petri nets (PEP) [30]. The main similarity between the two tools is that both deal with PBC and M-net models. The main difference is that PEP is oriented toward model-checking, proposing a graphical user interface to model Petri nets through various ways. The Petri nets models in PEP are fixed, mainly variants of PBC and a restricted version of M-nets, which cannot be changed by users. PEP also appears
to be not maintained anymore, the last release being dated of September 2004. PEP was the tool we used before to decide to develop SNAKES. The main reason for this decision was the impossibility to update PEP quickly enough with respect to theoretical developments in the PBC family, which is not surprising since PEP was never designed with this goal in mind. Like the PNK, PEP is released under GNU GPL.

Compared with CPN tools [14], SNAKES shares the ability to use a programming language for nets inscriptions: a variant of ML for CPN tools, and Python for SNAKES. This makes it possible to extend a lot the features of CPN tools. However, it uses fixed (but very general) Petri net models and does not provide support to introduce another variant. So, there might be new models of Petri nets that cannot be represented using one of those provided by CPN tools, and thus for which the tools cannot be used. Another important difference is that CPN tools feature an advanced graphical user interface while SNAKES is a programming library. Finally, CPN tools is not open source.

More generally, since any Petri net modelled in SNAKES may be executed, it could be compared with any Petri net tool that can simulate nets or compute their reachability graph. However, executing nets is not the main purpose of SNAKES and it is not designed to do it efficiently (even if it may be efficient enough for simulation). This feature is only required because using SNAKES for prototyping may require executing Petri nets. For instance, when working on a new model of Petri nets equipped with unbounded integer variables [34], the fact that SNAKES could execute this new class of net was necessary to prototype our state graph construction (see section 3.1 for details).

1.3 Outlines

The two next sections present SNAKES: its core library and its plugin system. Then we show three typical uses of SNAKES: writing a plugin to support a new variant of Petri net, writing a compiler for the Petri net semantics of a specification language, and using SNAKES as a service from another program. We conclude the paper with a list of ongoing and future works.

2 Core library

SNAKES is organised as a hierarchy of modules:

- **snakes** is the top-level module and defines exceptions used throughout the library;
- **snakes.data** defines basic data types (e.g., multisets and substitutions) and data manipulation functions (e.g., cross product);
- **snakes.typing** defines a typing system that can be used to restrict tokens accepted by a place (see section 2.2);
- **snakes.nets** defines all the classes directly related to Petri nets: places, transitions, arcs, nets, markings, reachability graphs, etc. A simplified class diagram of this module is presented in top of figure 3. It also exposes all the API from the modules above;
- **snakes.plugins** is the root for all the extension modules of SNAKES.

The first four modules above (plus additional internal ones not listed here) form the core library of SNAKES which is described further in the rest of this section. (Plugin system will be described in the next section.)

SNAKES is designed so that it can represent Petri nets in a very general fashion:

- each transition has a guard that can be an arbitrary Python Boolean expression;
- each place has a type that can be an arbitrary Python Boolean function that is used to accept or refuse tokens;
- tokens may be arbitrary Python objects;
- input arcs (i.e., from places to transitions) can be labelled by values that can be arbitrary Python object (to consume a known value), variables (to bind a token to a variable name) or several of these objects (to consume several tokens). New kind of arcs may be added (e.g., read arcs are provided as a simple extension of existing arcs);
- output arcs (i.e., from transitions to places) can be labelled the same way as input arcs, moreover, they can be labelled by arbitrary Python expressions in order to compute new values to be produced;
Quickly prototyping Petri nets tools with SNAKES

A simple coloured Petri net.

Fig. 1.

- a Petri net with these annotations is fully executable, the transition rule being that of coloured nets: a binding of variables must be found such that there are enough tokens in input places and the guard of the transition is respected as well as the type of the output places.

More precisely, at any marking, each transition can compute its enabling bindings (also called its modes) as follows:

- each combination of the available tokens with the variables on the input arcs provides a possible binding of these variables;
- each such binding corresponds to a Python environment (i.e., a set of names associated to values) in which the guard of the transition is evaluated;
- if the guard evaluates to True, each output arc is then evaluated in the same environment and it is checked if the produced tokens are accepted by the corresponding places;
- if all these tests pass successfully, the binding is enabling.

One of these enabling bindings can be used to fire the transition, which follows the same process starting from the second step and ending by actually consuming and producing the adequate tokens.

2.1 Example

A simple example of a coloured Petri net is depicted in figure 1. In this example, place \( p_1 \) can be marked by any integer-valued token (it currently holds two such tokens) and place \( p_2 \) is restricted to non-negative integers. In order to build this net within SNAKES, one may run the following Python code:

```python
from snakes.nets import *
n = PetriNet('simple_net')
n.add_place(Place('p1', [-1, 2], tInteger))
n.add_place(Place('p2', [], tNatural))
n.add_transition(Transition('t', Expression('x>0')))
n.add_input('p1', 't', Variable('x'))
n.add_output('p2', 't', Expression('x+1'))
```

Line 1 imports the main module. It exposes in particular classes PetriNet, Place, Transition, Expression and Variable, and objects tInteger and tNatural. Then, a Petri net is created, being given the name “simple net”. Two places are added to it, each is an instance of class Place whose constructor expects the name of the place, a list of tokens for its marking and an optional constraint on accepted tokens (see section 2.2). Similarly, a transition is added, being given a name and an optional Boolean expression for its guard. Finally, two arcs are created: one input arc labelled by a variable and one output arc labelled by an expression.

At the end of this program, various objects have been created, which is summarised in figure 2. We see on this diagram that a new class appeared: instances of MultiSet (from module snakes.data)

![Objects diagram after executing line 7 of the program. Some links indicate the names of the attributes that hold the references.](image-url)
are used to represent places markings. Moreover, attributes `pre` and `post` of places and transitions are dictionaries whose keys are node names and whose values are arc annotations. For instance `pl.post["t"]` is variable `x`, which is denoted this way on the diagram (instead of depicting the dictionary objects).

In order to get the list of enabling bindings for the transition, one may use the following:

```python
# alternatively, one could use:
t = n.transition('t')
m = t.modes()
```

At this point, the list of `m` is list `[Substitution(x=2)]` because only binding `{x: 2}` enables `t`. Then, the transition may be fired with the first binding discovered (if `t` had no enabling binding, this last statement would result in an exception as `m` would be an empty list):

```python
t.fire(m[0])
```

A class `StateGraph` is provided in order to automate this process and to compute the reachability graph by executing all the possible transitions for all the possible modes at all the reached markings. The following code creates the `StateGraph` object, computes all the reachable markings and then iterates over the states in order to print their information (marking, successors and predecessors).

```python
g = StateGraph(n)
g.build()
for s in g:
    print("state", s, "is", g.net.get_marking())
    print("successors:", g.successors())
    print("predecessors:", g.predecessors())
```

Executing this code after line 7 above prints the following, where each arc in the marking graph is labelled by the corresponding transition and its mode at firing time:

```python
state 0 is Marking({'p1': MultiSet([2, -1])})
successors: {1: (Transition('t', Expression('x>0')), Substitution(x=2))}
predecessors: {}
state 1 is Marking({'p2': MultiSet([3]), 'p1': MultiSet([-1])})
successors: {}
predecessors: {0: (Transition('t', Expression('x>0')), Substitution(x=2))}
```

### 2.2 Other features

**Type system for places.** As seen above, places in a Petri net are given a type that is used to control accepted tokens. We have used types `tInteger` and `tNatural` from module `snakes.typing`. This module actually provides a more general type system that one can use to build complex type checkers for places. In this system, a type is understood as a set of values, a type checker being a test that decides whether a given value belongs to the type or not.

Several type constructors are provided in order to build basic types:

- "Instances(c)" builds a type whose elements are instances of class `c`.
- "OneOf(a, b, ...)" creates a type whose values are just those enumerated, i.e., `a`, `b`, etc.
- "Collection(container, items, min, max)" creates a type for collections of objects whose values are objects in type `container` (usually list, set or tuple) and contain at least `min` and at most `max` values accepted by type `items`. There is similarly a type constructor Mapping for dictionary-like objects.
- "Range(first, last, step)" returns a type that accepts all the values ranging from `first` (included) to `last` (excluded) by steps of `step`.
- "Greater(min)" accepts all the values greater than `min`. Similarly, there are type constructors Less, GreaterOrEqual and LessOrEqual.
- "CrossProduct(t1, t2, ...)" accepts tuples of values from the given types `t1`, `t2`, ...
- "TypeCheck(fun)" creates a type whose values are those for which function `fun` returns True. This allows to build a type from an arbitrary Boolean function.

On this basis, types may be combined using various sets operators: & (intersection), | (union), − (difference), − (disjoint union) and ~ (complement). For instance, module `snakes.typing` defines:

```python
tInteger = Instance(int)
tNatural = tInteger & GreaterOrEqual(0)
```
A arcs. We have seen so far that arcs may be labelled by values, variables or expressions (only on output arcs). It is also possible to create multi-arcs that transport multiple values. For instance, \texttt{MultiArc([Value(1), Variable('x')])} can be the label of an input arc which is able to consume two tokens, one being value 1 and the other being an arbitrary value bound to variable \(x\).

SNAKES also provides test arcs that never transport values. On an input arc, this corresponds to a read arc; on an output arc, it is used to check the type of a place with respect to the annotation. For instance, creating an output arc with label \texttt{Test(Expression('x**2'))} will never produce a token in the corresponding place but will allow the transition to check if the place type accepts the value computed from the expression.

New kind of arcs may be created, it is only necessary to derive a class from abstract class \texttt{ArcAnnotation}. Moreover, like \texttt{Test} or \texttt{MultiArc} the new class may encapsulate existing arc classes.

Support for the Petri Net Markup Language. Every object in SNAKES can be exported to or imported from PNML. SNAKES provides a function \texttt{dump} that takes an object as argument and returns its representation in PNML. It also provides a function \texttt{load} that does the reverse, i.e., building an object from its PNML representation.

The PNML standard is still quite unstable with respect to many extensions of Petri nets, in particular coloured Petri nets. So, the implementation of PNML import/export in SNAKES has been made very flexible in order to allow easy updates when new standards will be published. When an object has no standard PNML representation, SNAKES uses either its own XML representation for objects defined in SNAKES, or Python serialisation (embedded in XML) for unknown objects. This is convenient because any object can be saved to PNML; but in such a case, there is few chance for the produced PNML to be compatible with an other tool. However, when a net can be considered has a place/transition Petri net, SNAKES reads and produces PNML that is conform to standards, which has been tested compatible with other Petri net tools that support PNML.

Controlling Python execution environment. It has been explained above how a transition binds variables on its input arcs in order to build an environment that is used to evaluate Python expressions in its guard and output arcs. When one of these expressions needs functions or modules that are not available in default environment, the evaluation fails. In such a case, it is necessary to declare the needed objects before to start executing transitions. There are two ways to do so.

One is to use method \texttt{declare} of a PetriNet instance that expects an arbitrary Python statement given as a string and executes it. If this statement has some side effects, this will be recorded for next evaluations. For instance, one may run \texttt{n.declare('import math')} in order to make module math available to all the evaluations occurring in net \(n\) after that.

The other solution is to access directly to the evaluation environment that is a dictionary stored as an attribute \texttt{globals}. This attribute exists for any object that needs to evaluate Python code and is shared over a whole Petri net. For instance, in order to declare a global variable \(x\), the method described above should be used as \(n\texttt{.declare('global.\_x\_x=2')}\) (first state that \(x\) is global and then assign it) or more simply, using attribute \texttt{globals}: \(n\texttt{.globals['\_x\_x']}=2\).

3 Extension modules

An extension module, or plugin, is meant to extend an existing module from the core library (usually \texttt{snakes\_nets}) by subclassing some of its classes or by defining new classes or functions. Because we do not know in advance which plugins will be loaded by users and in which order, classes hierarchy cannot be fixed statically. In order to do it dynamically, an extension module provides a function \texttt{extend} that takes as its single argument the module to extend, which may be \texttt{snakes\_nets} or a version of it already extended, and returns a new module with proper subclasses and auxiliary material. (Python allows to create classes at run time.) Module \texttt{snakes\_plugins} provides some functions to make this easy, allowing programmer to concentrate on writing the subclasses.

This approach is summarised in figure 3 that illustrate loading of plugin \texttt{gv} on the top of module \texttt{snakes\_nets}. First, \texttt{gv} depends on plugin \texttt{clusters}, so, \texttt{clusters} is loaded on the top of module \texttt{snakes\_nets}.

\footnote{There is no proper notion of abstract class in Python, however, this can be simulated using a class where methods that should be abstract raise \texttt{NotImplementedError} whenever called.}

45
which transparently calls `snakes.plugins.clusters.extend(snakes.nets)`. This returns a new module whose classes are those from `snakes.nets` except for `PetriNet` that come from `snakes.plugins.clusters`, with inheritance relations as depicted in figure 3. This new module also has class `Cluster` from plugin `clusters`. Then, on the top of the new module, `gv` is loaded, following a similar process. The result is a module that exposes the same interface as `snakes.nets` with two changes:

- its classes `PetriNet` and `StateGraph` come from `snakes.plugins.gv`;
- it has a new class `Cluster` that comes from `snakes.plugins.clusters`.

Loading of plugins is supported by helper functions, so, the example described above would simply reduces to the following code:

```python
import snakes.plugins
snakes.plugins.load('gv', 'snakes.nets', 'nets')
from nets import *
```

The first statement is a regular Python module import. The second statement is the loading of the plugin: the extension module called `gv` is loaded on the top of the module called `snakes.nets` and the result is imported as a module called `nets`. This allows to execute the last statement that imports every visible name from the newly constructed module `nets`, thus avoiding to use prefix “nets.” in order to access its content.

Several plugins may be loaded using a single `load` statement, in order to do so, it is just needed to give a list of plugin names to be loaded instead of only one. For instance:

```python
snakes.plugins.load(['gv', 'ops', 'synchro'], 'snakes.nets', 'nets')
```

### 3.1 Main available plugins

**Drawing nets and state graphs.** Plugin `gv` used above as illustration is dedicated to draw `PetriNet` and `StateGraph` objects using GraphViz [1] (see examples on figures 4 and 5). This plugin is a replacement of the former plugin `graphviz`. On the one hand, the new one introduces a dependency on Python module `pygraphviz` [23], but on the other hand, it provides much more control on drawing options and removes a call to an external program (which was both a security and a portability issue).
PBC and M-nets operations. The PBC and M-nets models define two kinds of operations, relying respectively on a labelling of places or transitions.

First comes a family of control flow compositions for which places are given statuses indicating their roles. In particular, one distinguishes entry places and exit places that correspond to the initial and final markings of a net. For instance, a sequential composition of two nets consists in combining the exit places of the first net with the entry places of the second net in such a way that the termination of the former corresponds to the starting of the latter. Moreover, statuses distinguish data places that are given a name, which corresponds to variables in a program. When nets are composed, data places with the same name are merged in order to ensure a unique representation of each variable.

Places statuses are implemented in plugin status, then plugin ops relies on it in order to implement the usual PBC control flow operations (with automatic merging of data places): sequence, iteration, choice and parallel. A name hiding operation is also provided, it removes the name of a data place so that it cannot be merged anymore, which corresponds to create a local variable. All these operations are implemented as Python operators, so, for instance, if n1 and n2 are two PetriNet objects, one may write:

```
1 p = n1 | n2  # parallel composition
2 s = n1 & n2  # sequential composition
3 c = n1 + n2  # choice
4 i = n1 * n2  # iteration
5 h = n1 / 'var'  # hiding of name 'var'
```

The other set of operations comprises transitions synchronisation, restriction and scoping. They are inspired from CCS [29] action synchronisation since transitions can synchronise on conjugated actions. However, with respect to CCS, PBC and M-nets use multi-actions, allowing more than two transitions to synchronise at the same time. Moreover, these models distinguish synchronisation that enables synchronised behaviour, from restriction that forbids independent behaviour (scoping is the successive application of both operations). Finally, these operations are purely statical and construct explicitly synchronised transitions that may fire or not at execution time. With respect to PBC, M-nets also define parameters for actions, allowing exchange of information between synchronised transitions.

This aspect is implemented in plugin synchr that defines classes for actions and multi-actions and adds methods to class PetriNet in order to perform the corresponding operations. SNAKES generalises the models by not imposing a fixed number of parameters for each action name. Instead, matching the number of parameters becomes a part of the unification process that takes place when two conjugated actions participate in a synchronisation.

The M-nets model also includes a general refinement operation that allows to replace a transition with an arbitrary M-net [16]. This operation has not been implemented in SNAKES and it is not intended to add it since it leads to very complex nets that are not tractable in practice. For instance, place types, tokens and arc annotations become trees after a refinement, so firing a transition implies matching trees against trees. Moreover, the refinement is always used for two purposes: synthesis of control flow operations (sequence, choice, etc.), and colour-safe execution of multiple instances of the same net. With new models of the family, both these effects are now feasible without using the general refinement, see [33] for instance.

Handling unbounded counters. Plugin lashdata has been developed while working on a model of p/T Petri nets equipped with unbounded integer variables [34]. This model has been given a semantics in terms of compact state graphs where one abstract state encodes possibly infinitely many concrete states that differ only by the values of the integers variables.

The fact that SNAKES was available and allowed to implement this model has been a benefit at several points. First, working on theoretical definitions and implementing them in parallel helped a lot to clarify and simplify definitions. Then, possible optimisations were discovered during implementation. Indeed, going to detailed program level allowed to identify where programming choices had to be made and thus to investigate consequences of different choices. Moreover, running the prototype on various examples allowed to exhibit cases where the current construction were not satisfactory, which led to improve the algorithm at theoretical level. Another source of satisfaction was the ability to produce automatically examples for illustrating the paper and the presentation, which was not only convenient but also increased confidence that no mistake was introduced in examples. Finally, the fact that a prototype existed for the construction described in the paper was perceived as very
positive by the referees as well as by the audience when the paper has been presented. An other work [25] about verification of multi-threaded systems modelled by coloured Petri nets has provided a similar experience.

Plugin lashdata relies on library Lash [8] to represent integers variables added to a Petri net. (With respect to [34], this plugin allows for any kind of Petri net and not only for p/t nets.) The plugin defines a class Data that encapsulates the data structures of Lash in order to store the values of the variables. For instance, the creation of a Petri net equipped with two variables x and y initialised to zero requires the following Python code:

```python
import snakes.plugins
snakes.plugins.load('lashdata', 'snakes.nets', 'nets')
from nets import *
n = PetriNet('N', lash=Data(x=0, y=0))
```

Then, the plugin extends transitions in order to take the integer variables into account: firing is subject to a condition on the variables (which is independent of the guard) and it is allowed to update their values. For instance one may write:

```python
n.add_transition(Transition('t'),
    condition='x<y',
    update='x=x+1; y=y-1')
```

Finally, class StateGraph is extended in order to implement the construction defined in the paper: several options are added to the constructor in order to enable various levels of compression. With no option, no compression is performed and the abstract state graph corresponds to the concrete reachability graph. Using option “loops=True” enables compression when a side-loop is detected (i.e., a transition that changes the variables but not the marking). Using “cycles=True” also enables compression when general loops are detected (i.e., cycles in the marking graph). Using “remove=True” then enables the removing of covered states (i.e., existing abstract states that are included in newly computed ones). Finally, using “fold=True” adds additional compression when sequences of the same transition are detected. This last option is not described in [34] and was added to the plugin after publication. Figure 4 shows an example of a state graph at different levels of compression.

**Fig. 4.** On top: a concrete marking graph. Below: its compact versions when the various compression options are activated (top-down: loops, cycles or remove, and fold). On this example, option remove does not provide more compression than cycles.
One hidden but important task of plugin lashdata is to perform the translation between the variable-based representation of data in Petri nets and its vector-based implementation in Lash. Indeed, Lash actually handles sets of integer-vectors and matrix-based linear conditions and updates. The plugin thus assigns to each variable an index in such a vector and computes for each condition or update expressed in Python (with syntactical constraints in order to ensure linearity) the corresponding matrices as expected by Lash. This work is facilitated by module snakes.compyler that was designed to handle Python code in SNAKES. For instance, it is used to correctly rename variables in expressions (e.g., in guards); it may be used also to translate Python expression to another language like C. More details about snakes.compyler is given in section 5 below.

4 Defining new Petri net variants

This section intends to illustrate how flexible is SNAKES with respect to the class of Petri net. For this purpose, support for a new class of Petri nets will be added using a plugin. We consider Merlin & Farber's time Petri nets [28]. A plugin is being developed to support them in SNAKES. In its current state, it extends classes Transition, Place, and PetriNet. In order to simplify the presentation, the version presented here is simplified, in particular it does not support transitions with multiple enabling, it is discussed below how this is supported in the actual implementation.

First, each transition is given an earliest and latest firing time as well as a timer (i.e., its current time value). Constructor of class Transition adds an attribute time for the timer, and two attributes min_time and max_time initialised from arguments added to the constructor. The default value for max_time is None, which is considered as an infinite boundary. Then, method enabled (that checks whether a binding enables or not a transition) is redefined in order to take time into account. It accepts an additional optional argument untimed that allows to avoid checking time boundaries. Method copy that duplicates a transition is also redefined in order to properly copy timing information.

Next, class Place is extended so that whenever the marking of a place is changed, its successor transitions are examined in order to reset their timer if their enabling is changed. This implies to redefine four methods: add that adds tokens to a place, remove that removes tokens from a place, reset that replaces the marking of a place, and empty that removes all the tokens from a place. In all cases, when a transition is newly enabled its timer is reset to zero. When a transition becomes disabled by its input marking, its timer is set to None, which avoids to consider it when time passes. As a result, value None for a timer indicates that the transition is not enabled because of marking, but when the timer is not None, its value has to be compared with earliest and latest firing time of the transition to know if it is enabled or not. Thus, method Transition.enabled is written as follows:

```python
def enabled (self, binding, untimed=None=False) :
    if self.time is None :
        return False
    elif untimed :
        return super(Transition, self).enabled(binding)
    elif self.max_time is None :
        return (self.min_time <= self.time) and super(Transition, self).enabled(binding)
    else :
        return (self.min_time <= self.time <= self.max_time) \
            and super(Transition, self).enabled(binding)
```

The condition on line 2 checks if the timer is None, in which case the transition is disabled because of marking. Otherwise, the condition on line 4 checks whether the new argument untimed has been set to True when calling the method. If so, the enabling is tested using the method of parent class super(Transition, self), thus ignoring all timing information. Then, line 6 corresponds to the case where no latest firing time has been given and the else case when it has been given. The assumption that a transition is not enabled when its timer is None is safe because this value is set by the pre-places of each transition when their markings are changed. For instance, methods Place.add and Place.remove are programmed as follows:

```python
def add (self, tokens) :
    enabled = self._post_enabled()
    super(Place, self).add(tokens)
    for name in self.post :
```

49
if not enabled[name] :
    trans = self.net. transition(name)
    if len(trans.modes()) > 0 :
        trans.time = 0.0

def remove (self, tokens) :
    enabled = self._post_enabled()
    super(Place, self).remove(tokens)
    for name in self.post :
        if enabled[name] :
            trans = self.net. transition(name)
            if len(trans.modes()) == 0 :
                trans.time = None

Method _post_enabled returns a dictionary whose keys are the names of the transitions in the post-
set of a place, associated to Boolean values indicating whether each transition is currently enabled or
not (which is checked by comparing its timer to None). Then, after adding the tokens line 3, method
add checks if a transition that was disabled becomes enabled by the new marking, i.e., if at least one
mode can be found for it (line 7). If so, its timer is set to 0.0. Symmetrically, method remove checks
if a transition that was enabled becomes disabled, in which case its timer is set to None. In order to
lift the implementation to the case where transition can be multiply enabled, it is enough to store
one timer for each mode of each transition, which is an easy change with respect to the simplified
implementation presented here.

Finally, class PetriNet is given two new methods step and time. The former computes the maximal
delay that can pass until the enabling changes, either by enabling a transition (when its timer reaches
its min_time), or by requiring a transition to fire (when its timer reaches its max_time). The latter
can be used to let time pass, which corresponds to increase all the timers that are not None. This
method time expects a duration as argument and returns the actual duration that could be used.
For instance, if a user request an increasing of 1.0 but if after 0.4 time units a transition becomes
newly enabled, then the method will only increase time by 0.4 and return this value to inform user.
Similarly, time will never be increased enough to overcome the latest firing time of a transition. So,
when a transition has to fire because of time, any call to time or step will return 0.0, corresponding
to the fact that time cannot pass before the urgent transition is fired.

These features have been implemented in less than 100 lines of code, including support for tran-
sitions with multiple enabling as described above. In order to have a complete plugin, it will be
necessary to implement an extended StateGraph class to construct a valid state space for time Petri
nets (e.g., one of those described in [4]).

5 Building a compiler for ABCD

A compiler for ABCD formalism has been first introduced in snakes to support a course about formal
specifications and Petri nets semantics of programming languages. We present it here in order to
illustrate how one can use module snakes.compyler to easily define the syntax and semantics of a
formal language that may include fragment of Python code. snakes.compyler can be considered as an
internal class of snakes core library because it is not exposed to end-users who deal with Petri nets
only. However, it is worth knowing it is here and what it proposes because it can solve many concrete
problems for one who wants to build a Petri net tool.

The syntax of ABCD defines atomic actions, enclosed in square brackets, that involve access to
data in buffers and an optional condition. For instance, \([a(x), b(y), c(x+y)]\) if \(x < y\) is such an
action that performs the following atomically:

- consume a value in a buffer \(a\) and bind it to variable \(x\);
- test the presence of a value in a buffer \(b\) and bind it to \(y\);
- send the result of \(x+y\) in a buffer \(c\);
- all this is guarded by condition \(x < y\) and the process is blocked until the guard becomes true.

Two distinguished atomic actions that perform no data access are [True] that can be executed uncondi-
tionally and [False] that is a deadlocked process.

Processes can be composed using four binary control flow operators; “+” is a choice, “:” is a
sequence, “[“ is a parallel and “*” is an iteration (execute repeatedly first operand and exit by executing
Below is a simple example of a 1-producer/2-consumers system specification. The producer puts in a buffer “bag” the integers ranging from 0 to 9. To do so, it uses a counter “count” that is repeatedly incremented until it reaches value 10, which allows to exit the loop. The first consumer consumes only odd values from the buffer, the second one consumes only even values. Both never stop looping.

```python

# buffer of integers declared empty
def bag():
    return ()

# buffer of integers initialised with the single value 0
def count():
    return 0

# guard: if x < 10
[count-(x), count+(x + 1), bag+(x) if x < 10] * [count-(x) if x == 10]

# transition guard: if (x % 2) == 1
[False]

# transition guard: if (x % 2) == 0
[False]

prod | odd | even
```

The Petri net resulting from this specification is drawn in figure 5. It is interesting to note that parts of this ABCD specification are actually Python code and could be arbitrarily complex:

- initial values of buffers (“()” and “0”);
- buffer accesses parameters (“x” and “x+1”);
- actions guards (“x<10”, “(x%2)==1”, . . . ).

In order to handle these parts, the ABCD compiler relies on module snakes.compyler that provides a series of tools to manipulate Python code. The module proposes two parsers: one is based on the native Python parser that is included in standard library, the other is a PLY [3] based parser that has been programmed independently. The former is fast and does not require maintaining if Python evolves, the latter can be extended and reused with much more flexibility. We will see here how it can be reused to parse ABCD specifications. The other tools in snakes.compyler are a set of classes to browse and transform abstract syntax trees (AST) returned by any of the parsers, and an structure of AST for constructs not included in Python, like those of ABCD. To observe the effect of using these tools, let us look at the guards of transitions in figure 5: they are not exactly those written in the specification. The reason is that they have been parsed, and their AST has been converted back to

Fig. 5. Petri net semantics of 1-producer/2-consumers specification.
text using one of the transformer classes called AstPrinter. Similarly, there exist a class AstRenamer that extends AstPrinter and allows for correctly renaming variables.

Extending the PLY parser of Python requires two steps: first, extend the lexer, then the parser. Indeed, we first need to add two tokens that do not exist in Python: “?” to parse buffer accesses like “b?(y)”, and “net” to parse net definitions. To do so, only a few lines are necessary:

```
class Lexer (PlyLexer) :
    def t_QUESTION (self, t) :
        r"\?" # prefix 'r' indicates a "raw" string where '\' is a regular character
        return t
    def t_NET (self, t) :
        "net"
        return t
```

Then we extend the parser: we use a new start symbol and new rules, but reuse symbols from Python grammar when we need to parse Python code. A PLY parser is a class in which each method corresponds to a rule in the grammar, which is very convenient and easy to use. For instance, here are two rules for parsing an atomic action:

```
def p_abcd_action_true (self, toks) :
    "abcd_action:LSQB:abcd_access_list:RSQB"
    toks[0] = Tree( "action", toks[2], test=True, net=None, lineno=toks.lineno(1))
def p_abcd_action_if (self, toks) :
    "abcd_action:LSQB:abcd_access_list:IF:IF:RSQB"
    toks[0] = Tree("action", toks[2], test=toks[4], net=None, lineno=toks.lineno(1))
```

Tokens LSQB, RSQN and IF correspond respectively to left and right square brackets that enclose an action, and to keyword if. Symbol abcd_access_list is a non-terminal of ABCD grammar that is handled by another rule. Finally, symbol testlist is a non-terminal of Python grammar that corresponds to a Boolean condition. Both these rules build an AST for the recognised action. The full parser has 31 rules, including one to handle parsing errors (plus all the rules inherited from Python grammar).

The last component of the ABCD compiler is in charge to transform the AST from the parser into a Petri net, which is quite a simple task because the operators in the language directly correspond to those implemented at Petri net level. Additionally, this translation attaches many information to the produced Petri net in order to expose its hierarchical structure and relate it to the ABCD code from which it was built.

This translation component requires 165 lines of code. All together, including command line handling and such auxiliary parts, the ABCD compiler is 450 lines long. So, we feel that it is indeed quick and simple to build such a tool with SNAKES in hands.

6 Using SNAKES as a service

SNAKES can be used as a service, which allows any program to use its features without the need to be written in Python. To do so, SNAKES includes a simple server that waits queries formatted as XML messages and answers with XML messages too. This system can be seen as a simplified XML RPC mechanism. The communication is made over UDP, which reduces to the minimum the complexity of writing a client: each query or answer is transported in a single UDP datagram. This makes this system suitable for communication local to a computer between one client and one server, but it is not intended to work over the Internet or with multiple clients: indeed, no reliability layer is added over UDP and there is no support for session nor authentication. Format of queries and answers is as simple as possible, and it is PNML as much as possible when Petri net related information is exchanged.

The query language is very simple but yet powerful, and it is easy to extend it by adding methods to class Query that is defined in plugin query. Only four kind of queries are currently recognised: “set”, “get”, “del” and “call”.

The “set” query assigns a value to a name at server side; the value may be any Python object formatted in PNML (extensions recognised by SNAKES ), for instance, it may be a Petri net.

```
<?xml version="1.0" encoding="utf-8"?>
<pnml>
```
Quickly prototyping Petri nets tools with SNAKES

3

<query name="set">
<argument>
<object type="str">name</object>
</argument>
</query>
</pnml>

The “set” query assigns a value to a name.

<query name="get">
<argument>
<object type="str">name</object>
</argument>
</query>
</pnml>

The “get” query returns the value previously assigned to a name.

<query name="del">
<argument>
<object type="str">name</object>
</argument>
</query>
</pnml>

The “del” query removes a name from server’s memory.

<query name="call">
<argument>
<object type="str">method_or_function_name</object>
</argument>
<argument>
... any PNML formatted value ...
</argument>
... as many arguments as needed ...
</query>
</pnml>

Finally, the “call” query performs a function or method call at server side and returns to the client what the call returns.

Even if very simple, this scheme is powerful enough to allow great flexibility because queries can be arbitrarily nested. So it is easy for instance to assign the result of a computation by nesting a “call” inside a “set”. Similarly, it is possible to call a method of an object returned by another call. It is also possible to execute arbitrary Python statements using exec or eval built-in routines.

By using an appropriate combination of those basic queries, one may build a set of template complex queries with placeholders where changing data has to be used. For instance, it is easy to build a query that lists the enabled transitions of a net, the only part that need to change is the name of the net. So, running this query from the client point of view is just a question of inserting a name at the right position into an XML formatted text, sending the resulting string over UDP and parsing the XML that it gets back in order to extract the appropriate information.

Like queries, answers have a simple formatting. There is basically two answers, depending on whether the query could be successfully executed or not. In case of an error, the answer is:

<answer status="error" error="ExceptionName">Exception message</answer>
When there is no error, status is set to “ok”, and the answer may include data or not, depending on whether the execution of the query returned data or not. This is for instance an answer without data:

```xml
<?xml version="1.0" encoding="utf-8"?>
<pnml>
  <answer status="ok"/>
</pnml>
```

If some return value has to be passed to the client, it is encoded in PNML and nested in tag `<answer>`:

```xml
<?xml version="1.0" encoding="utf-8"?>
<pnml>
  <answer status="ok">
    ... PNML data ...
  </answer>
</pnml>
```

7 Future and ongoing works

**snakes** is still under active development and is still considered as a beta software at its current version (0.9.3). It is planned to release the first stable version (numbered 1.0) when the documentation will be complete as well as the unit tests.

The current documentation for **snakes** is composed of an API reference manual, a tutorial and a couple of text files. The former is automatically generated from comments in source code (Python “docstrings”) that also contains the code used for unit testing. There are currently some modules, classes, methods or functions that lack a proper documentation with sensible examples and unit test. This must be fixed in order to improve usability and increase the confidence about the absence of bugs. However, the core library is currently quite complete to this respect: 96% is documented, 87% has unit test and 95% has detailed API specification. The rest of the documentation is written separately; currently, the tutorial does not introduce all the plugins of **snakes**, but its is quite complete about the core library.

Support for other formats. Apart for PNML, other file formats should be supported by **snakes**. In particular, those of various model checkers that do not support PNML. The idea is that **snakes** can be used to build a model taking advantage of the PBC and M-nets compositions, then this model can be model-checked with a specialised tool. This work requires first a survey of the existing tools to see which ones do not support PNML and which formats are already supported by some conversion tool. The goal is to minimise the number of output formats to add to **snakes** in order to avoid to duplicate existing tools. For instance, PEP or the model-checking kit [18] provide quite a lot of such translation tools.

There may exist also formats for which it may be interesting to have an input filter to **snakes**, but it is likely that very few ones are worth implementing. Indeed, **snakes** is not intended to be used at the end of a tool chain but rather at the beginning.

Support for more Petri nets variants. From the beginning, **snakes** has been designed to be able to support as many variants of Petri nets as possible. This aim should be turned into acts by providing extension modules for popular Petri net models, in particular, those with time (see [13] for a comprehensive survey), stochastic Petri nets [2] and object Petri nets [21]. As presented above, Merlin & Farber’s time Petri nets can be implemented with no particular difficulty.

API to other programming languages. As every library, **snakes** is bound to a particular programming language, Python in its case. This may be a limitation to its usage, even if Python is a language that is very easy to learn. In particular, this is a limitation for a programmer that would like to integrate **snakes** with another application not written in Python. For instance, a graphical editor could use **snakes** as its data model in order to provide simulation capabilities. This need is partially addressed by the server mode of **snakes** but this solution involves many XML encoding and decoding; using **snakes** through an API would be much simpler and more efficient.
In order to remove this limitation, a C binding of snakes is being developed. By automatically inspecting the actual Python code, wrappers for each class, method and function can be generated as Pyrex [20] code. This is a dialect of Python mixed with C that can be compiled to pure C code. It is primarily intended for building Python extensions but is also suitable for embedding Python into C programs. A module called apix has been introduced recently to extract and explore snakes’ API. Exporting to Pyrex will be the next step.

Then, with a C API available, many other languages can be mapped. Thin bindings can be easily obtained for a variety of languages using a tool like SWIG [15]. But it is often more interesting to produce a thick binding that is oriented toward the programming style enforced by a particular language. For instance, a Java binding should be object-oriented as the original API is. (Note that there is no a priori limitation with Java that can be interfaced to native libraries using JNI framework [38].)

8 Getting involved

Anybody who wish to contribute to snakes is welcome. It is not necessary to produce Python code for snakes in order to make a useful contribution: experiment reports, bug reports, documentation, feature requests, translations, advertising, etc., are as important as source code. (Source code is obviously welcome too.)

In order to be notified of snakes’ releases, one can register at its FreshMeat page at ⟨http://freshmeat.net/projects/snakes⟩.

In order to report bugs, request features, ask questions or contribute code, one may use snakes’ LaunchPad page at ⟨https://launchpad.net/snakes⟩. This page also gives access to a Bazaar VCS repository where latest changes to snakes are made available in (almost) real-time.

9 Acknowledgements

I would like to thank Hanna Klaudel for her good advices about how to present this paper, and Emmanuel Polonowski who helped with UML notation. Let me also thank the organisers of PNTAP 2008 for having proposed this paper for publication into the Petri net newsletter.

References

1. AT&T Research. Graphviz, graph visualization software. ⟨http://www.graphviz.org⟩.
24. F. Heitmann. Petri nets tool database at the Petri net world. ⟨http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools⟩.
38. Sun Microsystems, Inc. Java native interface. ⟨http://java.sun.com/j2se/1.5.0/docs/guide/jni⟩.
PETRI NET NEWSLETTER

The rapid information service on Petri Nets and related system models published by the Special Interest Group on Petri Nets and related system models (FG 0.0.1) of the Gesellschaft für Informatik e.V. (GI)

Two issues per year. 66 issues have been published from 1979 to 2004.

Topics include:

- technical contributions
- state-of-the-art reports
- reports on new books
- information on software packages
- project announcements and reports
- conference announcements, programs, and reports
- abstracts of recent publications

Members of FG 0.0.1 of the GI receive the Newsletter free of charge. This membership causes no further obligations or charges. For becoming a member send this application form to

Gesellschaft für Informatik e.V.
Wissenschaftszentrum
Ahrstraße 45
D - 53175 Bonn
Germany
fax +49 228 302 167

Additional and former issues of the Newsletter can be obtained from the same address.

-----------------------------------

Application Form
Petri Net Newsletter
I order the Petri Net Newsletter starting from January, _____ until recalled, connected with membership in the special interest group FG 0.0.1 *Petri Nets and related system models* of the Gesellschaft für Informatik e.V.

The annual fee is:
- EURO 10,00 for members of GI
- EURO 15,00 for non-members of GI

<table>
<thead>
<tr>
<th>title/Mr/Ms</th>
<th>last name</th>
<th>first name/initials</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>address</th>
<th>O affiliation / O private</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>phone</th>
<th>fax</th>
<th>email</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

GI membership number (members only)

<table>
<thead>
<tr>
<th>place and date</th>
<th>signature</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Form of payment:**
Transfer by account:

*Accounts of Gesellschaft für Informatik e.V.:
  Post office account no. 1981 39502, Postbank Köln
  Bank account no. 46 581, Sparkasse Bonn (Bank code 38050000)

If a foreign bank is charged with the transfer, the invoice amount increases by EURO 8,00 bank fee.
Foreigners can also pay by credit card (not for German members).
Diners Club, Visa Card and Master Card are accepted.

<table>
<thead>
<tr>
<th>credit card company</th>
<th>card number</th>
<th>expiration date</th>
</tr>
</thead>
</table>