Debugging Non-Determinism: a Petrinets Modelling, Analysis, and Debugging Tool

(Tool Demonstration)

Simon Van Mierlo
University of Antwerp
Email: simon.vanmierlo@uantwerpen.be

Hans Vangheluwe
University of Antwerp
Flanders Make vzw
McGill University
Email: hv@cs.mcgill.ca

***Abstract*—Non-deterministic formalisms are used to model systems whose runtime behaviour is inherently non-deterministic (its runtime execution might be different in consecutive runs, even for the same inputs). To analyse these systems, the full state space is explored to check whether an unwanted state (for example: deadlock, unsafe) can be reached. Debugging support for such formalisms is currently limited. This paper presents a prototype tool which allows to interactively construct the reachability graph (by manually stepping). The construction can be automatically paused at the moment a state of interest is reached (breakpointing). This should lead to earlier detection of errors and easier resolution, since the user can observe and control the reachability analysis.**

I. Introduction

Non-deterministic formalisms, such as Petrinets [1] provide abstractions for natively modelling systems whose (inherent) runtime behaviour can differ from one execution to the next, even for the same inputs. Examples include, amongst others, distributed systems, safety-critical embedded systems that can be interrupted by its environment, and programs executing on multiple threads accessing shared resources. By modelling their inherent non-determinism, it becomes possible to exhaustively explore the state space of the system and infer properties concerning its safety and liveness. A full state space exploration might, however, take a long time to complete for non-trivial systems.

Recently, multiple works have introduced debugging operations for a number of modelling formalisms [2], [3], [4]. Often, these techniques deal with deterministic formalisms whose semantics result in an execution trace that evolves the state of the system over time. Common operations, transposed from code debugging, include pausing, stepping (at several levels of granularity), (scaled) real-time simulation and breakpoints (automatic pauses on state conditions).

Debugging operations for non-deterministic formalisms have not been thoroughly researched. This paper presents a prototype debugging tool for Petrinets, which is a basic, but representative example of a non-deterministic formalism. Usually, to analyse a Petrinets model, its reachability graph is constructed and subsequently queried: if an undesirable state can be reached (for example, a deadlock state), the user needs to manually check how that invalid state was reached. Tools for analysing Petrinet models offer limited debugging capabilities. As an example, TINA [5] allows to step through a model's firing sequence manually, but does not offer more advanced capabilities such as breakpoints.

We extend the set of existing analysis operations with interactive debugging operations that allow to steer the reachability analysis in a direction deemed most useful by the modeller. In a sense, we envision to combine the existing formal analysis techniques with debugging operations to get the best of both worlds: the reachability graph is constructed up to a point deemed interesting by the modeller (and paused automatically using a breakpoint), after which they are free to step through the rest of the reachability analysis by hand. While our tool is built for the Petrinets formalism, the techniques are general and can be applied to other non-deterministic formalisms.

II. Tool Overview

This section describes the Petrinets debugger. We first look at its architecture, which is based on the Modelverse [6], our (meta)modelling kernel and repository. Then, we describe the visual interface used for modelling Petrinet models and debugging their reachability graphs.

A. Architecture

Our architecture is based on the Modelverse, our modelling framework and repository [6]. In the Modelverse, we model the abstract syntax of the Petrinets language: a model in the language consists of a number of *Place*s and *Transition*s. Places can have a number of (initial) *tokens* (an integer number) and can be connected to transitions; transitions can be connected to places as well.

The semantics of the language is defined in the Modelverse in the form of an action language model (an analyser). The analyser produces a reachability graph, which consists of a set of reachable markings. A marking contains a number of tokens for each place. These markings are connected: a link denotes that the destination marking can be reached from the source marking when a transition is triggered. A transition can be triggered when all of its input places contain at least

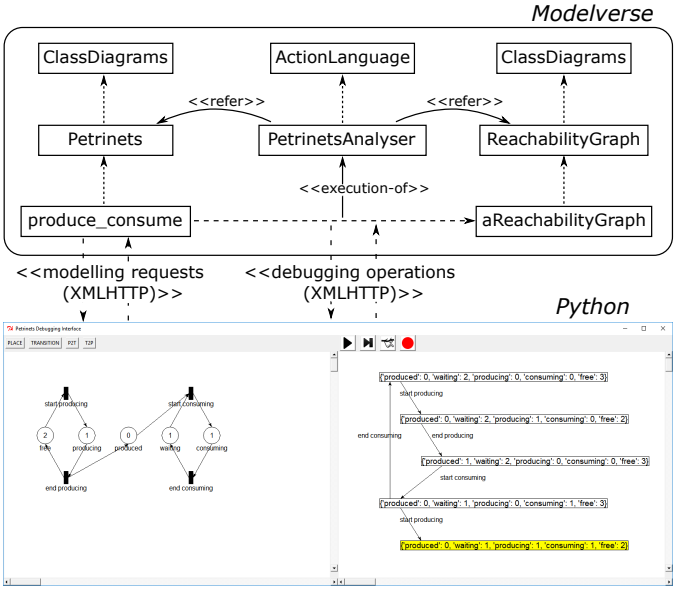

Fig. 1. The architecture of our solution.

one token; it then subtracts one token from all its input places and adds a token in all its output places (resulting in a new reachable marking).

Figure 1 visualises the architecture of our solution: the Modelverse provides all modelling operations and allows to define an analyser in action code. It is deployed on a server which provides an API that is called using XMLHTTP requests. For visualization purposes, we implement a Python Tkinter application which allows to visually model Petrinets in one window and analyse/debug them, showing the reachability graph in a separate frame and allowing the user to interact with it. The following subsection explains which debugging operations we added to the reachability graph construction algorithm. Details on the visual modelling and debugging interface are provided in the last subsection.

B. Debugging Operations

We define the following debugging operations:

- **Continuous Operation** executes the reachability graph construction algorithm until the full reachability graph has been generated (this is equivalent to running the analyser without debugging support).
- **Step** executes one analysis "step": it generates one additional reachable node in the reachability graph.
- **Manual** allows the user to fully control the analyser through the visual interface. The debuggable analyser will expose these phases in the algorithm and allow the user to control them:
  1) The analyser communicates all markings that are "unexplored". An unexplored marking has at least one enabled, non-explored transition. Unexplored markings are highlighted in the interface, and the user can choose on of them.
  2) When the user has chosen one of the unexplored markings, the marking is loaded into the Petrinet model and the analyser communicates which transitions are enabled (and unexplored) in that particular marking. These transitions are highlighted in green and the user can choose one of them by clicking on it.
  3) The analyser executes the transition and updates the reachability graph.

  This mode is most interesting to go back to a (partially) unexplored node in the reachability graph and continue its construction from that point interactively.
- **Breakpoints** allow the user to automatically pause the analysis when a certain condition on the state (*i.e.*, the structure of the reachability graph) is reached. From there, the user can use one of the above operations to resume construction.

C. Interface

The visual modelling, analysis, and debugging interface is shown in Figure 2. At the top, the modelling interface for Petrinets is shown with a loaded model that describes a producer-consumer system **(1)**. A toolbar allows to edit the model **(2)**. Below, the (partial) reachability graph is shown **(3)**. A toolbar allows the user to control the analysis using the operations defined in the previous subsection **(4)**. In the screenshot, the user is manually controlling the analysis algorithm and has selected the bottom-most marking (highlighted in yellow). In the Petrinets model, the marking is loaded in the places, and the enabled transitions are highlighted in green. By clicking one of them, a new (unexplored) marking will be added to the reachability graph.

III. Conclusion

This paper presents a prototype visual modelling, analysis and debugging interface for Petrinets, chosen as a representative non-deterministic formalism. The interface allows to model Petrinets and analyse them, visualising its reachability graph as it is constructed. Additional control is given to the user in the form of debugging operations: the user can step through the reachability graph construction and manually control which marking is generated next. This allows users to interactively explore the reachability graph. In future work, these techniques can be used to build debuggers for more advanced non-deterministic formalisms (such as model transformations [7], process modelling formalisms, etc.).

Acknowledgment

This work was funded by Flanders Make vzw, the strategic research centre for the manufacturing industry, and with a PhD fellowship from the Agency for Innovation by Science and Technology in Flanders (IWT). We thank Yentl Van Tendeloo for his support of and with the Modelverse.

References

[1] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, Apr 1989.

[2] E. Bousse, J. Corley, B. Combemale, J. Gray, and B. Baudry, "Supporting efficient and advanced omniscient debugging for xdsmls," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2015. New York, NY, USA: ACM, 2015, pp. 137–148.

[3] A. Chiş, M. Denker, T. Gîrba, and O. Nierstrasz, "Practical domain-specific debuggers using the moldable debugger framework," *Comput. Lang. Syst. Struct.*, vol. 44, no. PA, pp. 89–113, Dec. 2015.

[4] S. Van Mierlo, Y. Van Tendeloo, and H. Vangheluwe, "Debugging Parallel DEVS," *SIMULATION*, vol. 93, no. 4, pp. 285–306, 2017. [Online]. Available: http://dx.doi.org/10.1177/0037549716658360

[5] B. Berthomieu and F. Vernadat, "Time petri nets analysis with tina," in *Proceedings of the 3rd International Conference on the Quantitative Evaluation of Systems*, ser. QEST '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 123–124. [Online]. Available: http://dx.doi.org/10.1109/QEST.2006.56

[6] Y. Van Tendeloo, "Foundations of a multi-paradigm modelling tool," in *MoDELS ACM Student Research Competition*, 2015, pp. 52–57.

[7] J. Corley, B. P. Eddy, E. Syriani, and J. Gray, "Efficient and scalable omniscient debugging for model transformations," *Software Quality Journal*, pp. 1–42, 2016.

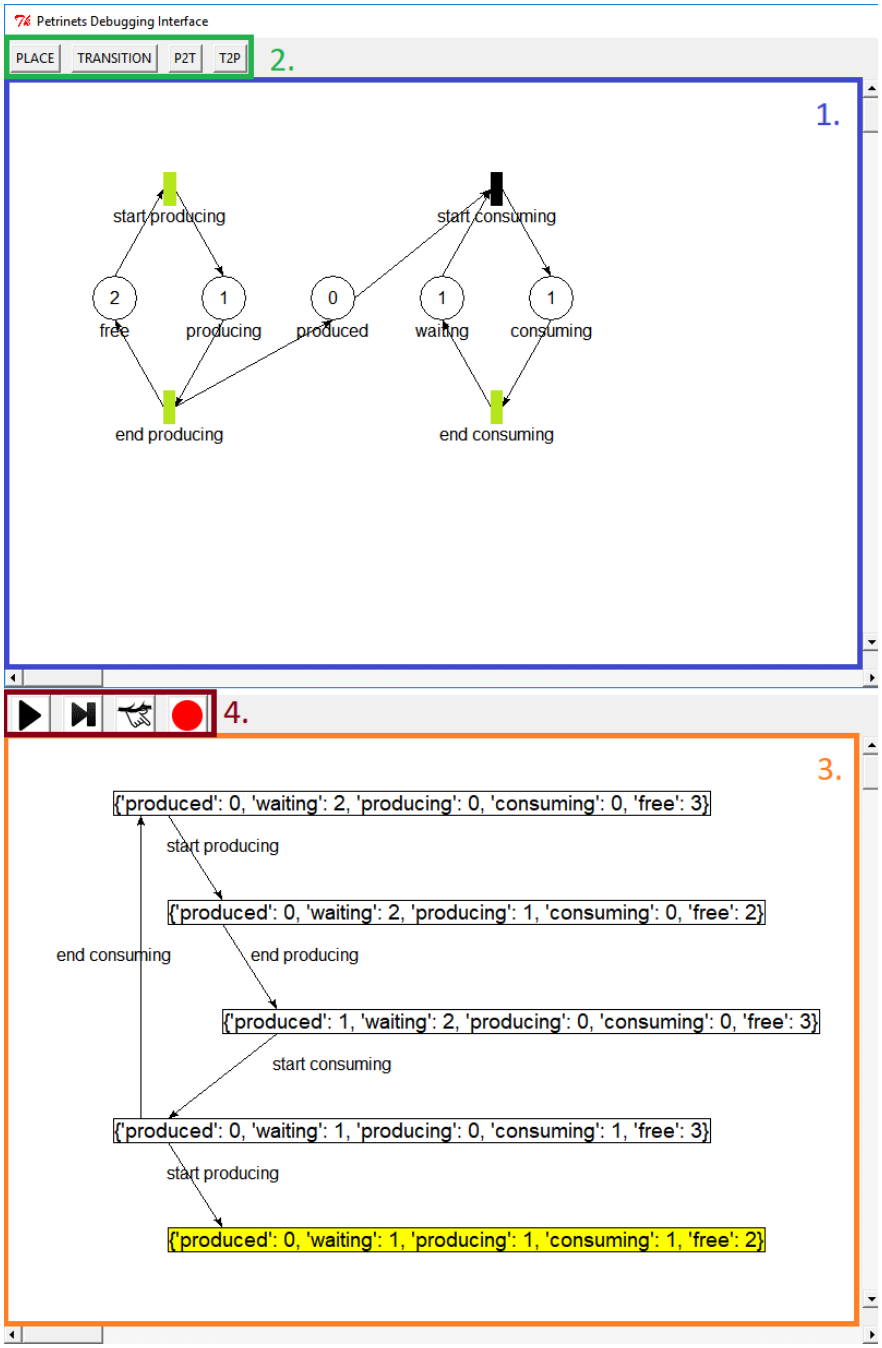

Fig. 2. The visual modelling and debugging interface.