

Towards a Rewriting-Logic Semantics of P

Francisco Durán¹, Nicolás Pozas¹, Carlos Ramírez², and Camilo Rocha²

¹ Universidad de Málaga, Málaga, Spain

² Pontificia Universidad Javeriana, Cali, Colombia

Abstract. P is a programming language based on state machines, suited for the definition of asynchronous event-driven *open* systems. The language was designed with verification in mind and it is bundled with back-end analysis engines for, e.g., bounded model checking. With the aim of extending such verification capabilities, a semantics of P in rewriting logic has been developed. Given this formal semantics, developed using the rewriting-logic language Maude, the Maude Formal Environment (MFE) can now be used to carry on different types of analyses on P programs.

Keywords: P · Rewriting logic · Maude

1 Introduction

P [6]³ is a domain-specific language designed for the specification of asynchronous event-driven systems. In P, a system is specified as a collection of state machines that use events to communicate. The underlying model of computation for P programs is communicating state machines (or actors) [9,1]. P programs can be automatically compiled into executable code, which can then be used in combination with state exploration techniques to identify and correct design bugs.

The approach presents well-known advantages. The model can be recompiled as many times as needed, and the maintenance of the implementation directly happens on the state machine diagrams. Similarly, the properties checked on the state machine diagrams will also be true on the code automatically generated from them. P has been used to implement and verify several applications (see P's web site in Footnote 3). Among these, the implementation and verification of the core of the USB device driver stack in Microsoft Windows 8, the formal verification of the core distributed protocols in the strong consistency launch of Amazon S3, and the formal specification and verification of the OTA protocol in AWS FreeRTOS.⁴

The design of the P language ensures that a P program can be checked to guarantee its responsiveness, i.e., its capacity to handle all events in a timely

³ P's web site: <https://p-org.github.io/P/>.

⁴ The web site of Amazon's real-time operating system for resource-constrained devices (FreeRTOS) is at <https://aws.amazon.com/freertos/>.

manner. Of course, to be able to model complex distributed systems, the language provides the possibility of specifying different types of machines, states, actions, events, etc. For instance, to enable the delayed processing of specific events in certain states, the language provides *deferred* events. For testing purposes, the programmer can define the environment in which the system is to be executed by creating nondeterministic *ghost* machines, which are later eliminated during the compilation process.

In P, a program is comprised of multiple state machines that communicate through events. Each machine has a set of states, actions, and local variables. The states and actions contain code statements for reading and updating local variables, sending events to other machines, raising local events, or invoking external C functions, which are mainly intended for the handling of data transfer. When a machine receives an event, it executes transitions and actions, causing the aforementioned code fragments to run. For programming convenience, auxiliary functions can be used.

In practice, handling every event at every state leads to combinatorial explosion in the number of control states, making impractical the formal verification of many interesting case studies. To alleviate this, in addition to its compiler, P provides a tool for explicit-state bounded model checking. Several different model checkers have been used associated to different versions of P. Initially an ad-hoc tool was developed as P companion [6], then Zing [2] was used, and then Coyote [5]. Version 2 of P is again relying on an ad-hoc tool, which is basically a simpler version of Coyote, in which some of the developers of P were also involved (see Footnote 3).

The aim of this work is to complement the set of tools already available for the P language with reachability analysis and more comprehensive model checking. Given a rewriting logic semantics of P in Maude [4], the Maude toolset can be used to carry on a new variety of verification techniques. The combinatorial explosion can then be tackled using different or a combination of techniques. Equational abstraction [10,4] may be the simplest one. But once the Maude specification is available, it also opens up the possibility of performing different forms of symbolic analysis [3,7,8].

Some of these ideas are illustrated with the help of a simple example. Figures 1 and 2 show the state machines of a dice and a simple ghost machine exercising the dice. Once the dice machine is created, the ghost machine keeps sending throw events to it. On the other hand, after some initialization, the dice machine keeps track of the number of times each face shows. The state machines in the figures use the standard notation, which is self-explanatory. Notice that the ghost machine keeps a reference to the dice it creates and that the dice keeps the frequency of each face in a map. Even though the figures show the corresponding graphical representation of the machines, there is a textual representation of P programs that is processed by the Maude specification.

Without diving into unnecessary detail, a machine is modeled as an actor, with a unique identifier, a type, and a(n unbounded) queue of incoming events. Under execution, the representation of a machine also keeps its current state,

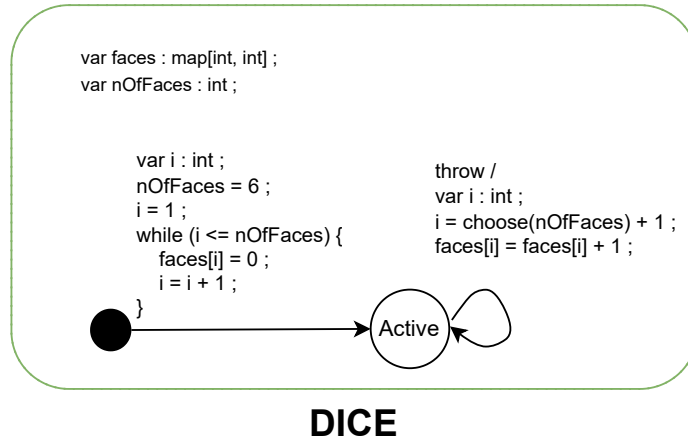


Fig. 1: The Dice state machine

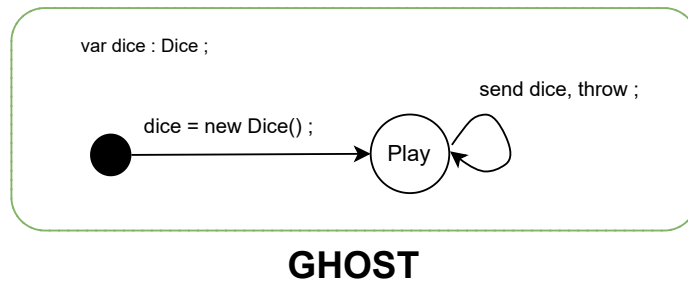


Fig. 2: The Ghost state machine

the code currently being executed, and its memory stack. Function declarations complete the definition of a machine. A system is then represented as a term composed of a set of executing machines, an index for the generation of fresh names, sets of declarations of machines, global functions, and tests, and a string to collect the log of the execution.

Given the appropriate declarations defining the structures of P programs as a membership-equational-theory (Σ, E) , the operational semantics is then given as a set of transformation rules defining a rewrite theory $\mathcal{R} = (\Sigma, E, R)$. In this case, the interest lies in exploring all possibilities and checking properties against them. For example, the search command can be used to check for reachable states satisfying certain condition (e.g., falsifying certain invariant).

Model checking cannot be used directly on this example, since its state space is infinite. An abstraction can be made to make it finite. Depending on the property to be checked, the abstraction will be defined in one way or another. On the abstracted system, the property of interest can now be verified.

Acknowledgments. This work has been partially supported by projects TED2021-130666B-I00 and PID2021-125527NB-I00, and an Amazon Research Award.

References

1. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. PhD thesis, MIT, 1986.
2. T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. In R. Alur and D. A. Peled, editors, *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 484–487. Springer, 2004.
3. K. Bae and J. Meseguer. Predicate abstraction of rewrite theories. In G. Dowek, editor, *Rewriting and Typed Lambda Calculi - Joint International Conference, RTA-TLCA 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8560 of *Lecture Notes in Computer Science*, pages 61–76. Springer, 2014.
4. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.
5. P. Deligiannis, A. Senthilnathan, F. Nayyar, C. Lovett, and A. Lal. Industrial-strength controlled concurrency testing for sc C tt # programs with sc coyote. In S. Sankaranarayanan and N. Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings, Part II*, volume 13994 of *Lecture Notes in Computer Science*, pages 433–452. Springer, 2023.
6. A. Desai, V. Gupta, E. K. Jackson, S. Qadeer, S. K. Rajamani, and D. Zufferey. P: safe asynchronous event-driven programming. In H. Boehm and C. Flanagan,

- editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 321–332. ACM, 2013.
7. F. Durán, S. Eker, S. Escobar, N. Martí-Oliet, J. Meseguer, R. Rubio, and C. L. Talcott. Programming and symbolic computation in Maude. *J. Log. Algebraic Methods Program.*, 110, 2020.
 8. F. Durán, S. Eker, S. Escobar, N. Martí-Oliet, J. Meseguer, R. Rubio, and C. L. Talcott. Equational unification and matching, and symbolic reachability analysis in Maude 3.2 (system description). In J. Blanchette, L. Kovács, and D. Pattinson, editors, *Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings*, volume 13385 of *Lecture Notes in Computer Science*, pages 529–540. Springer, 2022.
 9. C. Hewitt, P. B. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In N. J. Nilsson, editor, *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Stanford, CA, USA, August 20-23, 1973*, pages 235–245. William Kaufmann, 1973.
 10. J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. *Theor. Comput. Sci.*, 403(2-3):239–264, 2008.