jpf-ctl: CTL Model Checking of Java Code

Parssa Khazra, Anto Nanah Ji, Matt Walker, Hongru Wang, and Franck van Breugel
Department of Electrical Engineering and Computer Science, York University, Toronto

Although several attempts have been made to extend *Java PathFinder* (JPF) [7] to support model checking of linear temporal logic (LTL) [6], we are not aware of any extension of JPF that provides *computation tree logic* (CTL) [3] model checking. Our extension, named jpf-ctl, extends JPF so it can check whether a Java app satisfies a property expressed in CTL. An overview of our tool can be found in Figure 1. In the remainder of this abstract, we briefly discuss the four components that we developed.

Compiler By means of ANTLR [5], we generated a lexer and a parser. We implemented an inheritance hierarchy of classes to represent the abstract syntax trees. We used ANTLR's support of the visitor design pattern to transform the parse tree generated by the lexer and parser into an abstract syntax tree.

Converter The converter extracts from the abstract syntax tree the atomic propositions that are used in the CTL formula. Currently, our tool only supports static boolean fields as atomic propositions. The converter combines this information with other configuration data, such as JPF's classpath and native classpath, into an application properties file that is used by JPF.

Listener Our listener generates a *partial* transition system that models the Java code. Such a partial transition system differs from an ordinary transition system in that some states may be designated as partially explored. States may only be partially explored as JPF runs out of memory or takes too much time. The extension jpf-label [4] allows us to label the states with atomic propositions.

Model checker Given a labelled partial transition system and the abstract syntax tree of a CTL formula, we implemented a model checking algorithm to determine whether the system satisfies the formula. Like the algorithm presented in [2] to deal with partial systems, our algorithm uses a three-valued logic and computes a pessimistic interpretation and an optimistic interpretation. Given a CTL formula, we approximate the set of states of the partial system that satisfy the formula no matter how the partial system is extended. Furthermore, we also approximate the set of states of the partial system for which there exists an extension that satisfies the formula. To compute these pessimistic and optimistic approximations of the satisfaction set of a formula, we use variations on the standard algorithm to compute the satisfaction set of a formula (see, for example, [1, Section 6.4]) and prove our algorithm correct.

Given a CTL formula φ, let us denote the pessimistic approximation of the satisfaction set of φ by P_φ and the optimistic one by O_φ. If the initial state of the labelled partial transition system belongs to P_φ then we can conclude that the system satisfies φ. In this case, the result of the model checker is a witness that illustrates that the partial system satisfies φ. If the initial state of the labelled partial transition system does not belong to O_φ then we can conclude that the system does not satisfy φ. In this case, the result of the model checker is a counterexample that illustrates that the partial system does not satisfy φ. Otherwise, the initial state of the labelled partial transition system does not belong to P_φ but belongs to O_φ, since $P_\varphi \subseteq O_\varphi$. This third value corresponds to the case in which we cannot conclude whether the partial system satisfies φ.

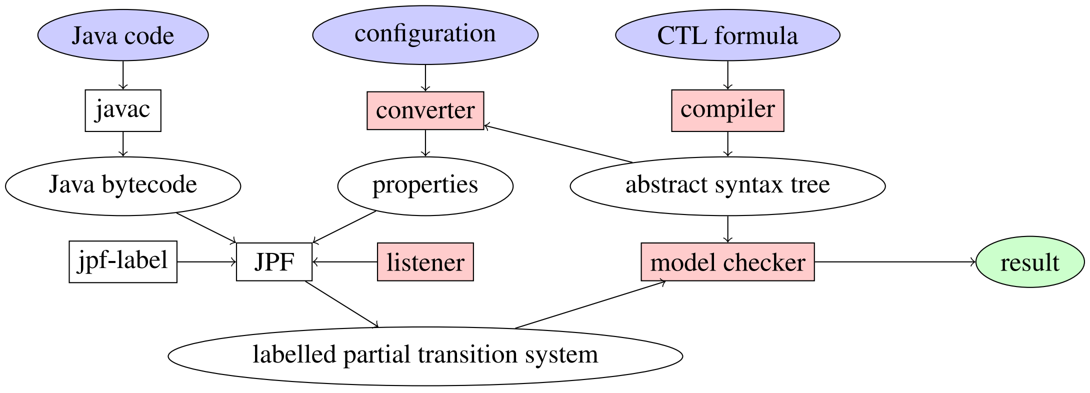

Figure 1: The diagram provides an overview of the CTL model checking tool. The ovals are data and the rectangles are tools. The blue ovals are input. The green ovals are output. The red rectangles are the parts that we developed.

References

[1] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, Cambridge, MA, USA, 2008.

[2] Glenn Bruns and Patrice Godefroid. Model checking partial state spaces with 3-valued temporal logics. In Nicolas Halbwachs and Doron Peled, editors, *Proceedings of the 11th International Conference on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, Trento, Italy, July 1999. Springer-Verlag.

[3] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Proceedings of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, NY, USA, May 1981. Springer-Verlag.

[4] Syyeda Zainab Fatmi. Probabilistic model checking of randomized Java code. Master's thesis, York University, Toronto, Canada, September 2020.

[5] Terence Parr. *The Definitive ANTLR 4 Reference*. The Pragmatic Bookshelf, Dallas, TX, USA, 2013.

[6] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Providence, RI, USA, October/November 1977. IEEE.

[7] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, April 2003.