PreDiaG: A Tool for the Generation of Predicate Diagrams

Cecilia E. Nugraheni

Dept. of Computer Science, University of Munich, Germany
nugrahen@informatik.uni-muenchen.de

Abstract. This paper presents PreDiaG (Predicate Diagrams Generator), a tool prototype for the verification of reactive systems using predicate diagrams, which are a class of diagrams that represent the abstractions of reactive systems. PreDiaG supports the generation of predicate diagrams automatically based on the algorithm in [3]. Given three inputs, namely a system specification written in a subset of TLA$^+$ [12], a set of state predicates and a set of abstraction function and rewriting rules, this tool generates the suitable predicate diagram using the concept of abstract interpretation. The generated diagram can be encoded in Promela, the input language of model checker SPIN [9].

KEYWORDS: reactive systems, abstraction, verification, predicate diagrams, tool, TLA, SPIN.

1 Introduction

Abstraction is an attractive method for proving a temporal property for a reactive system. Given a reactive system $Spec$ and a temporal property F, the proof that $Spec$ satisfies F can be done by finding a simpler abstract system $\overline{Spec}$ such that if $\overline{Spec}$ satisfies F, then $Spec$ satisfies F as well [5].

Predicate diagrams, first introduced in [4], are a class of diagrams that represent abstractions of reactive systems described by specification written in temporal logic. These diagrams are intended as the basis for the verification of both safety and liveness properties of reactive systems.

In this framework the specifications of reactive systems are written as TLA$^+$ formulas [12] of the form: $Spec \equiv Init \wedge \Box[Next]_v \wedge L$ where $Init$ is a state predicate that characterizes the system's initial state, $Next$ is an action formula representing the next-state relation, v is the tuple of state variables of interest and L is a conjunction of formulas $\mathrm{WF}_v(A)$ and $\mathrm{SF}_v(A)$, where $\mathrm{WF}_v(A)$ and $\mathrm{SF}_v(A)$ are formulas that assert weak and strong fairness condition for the action formula $\langle A \rangle_v$ respectively.

A predicate diagram G over a set of state predicates $\mathcal{P}$ and a set of action formulas $\mathcal{A}$ is a finite graph whose nodes are labeled with sets of (possible negated) predicates in $\mathcal{P}$, and whose edges are labeled with action names from the action formula in $\mathcal{A}$ and possibly with an ordering annotation. Every action in $\mathcal{A}$ may

Mária Bieliková (Ed.): SOFSEM 2002 Student Research Forum, pp. 35–40, 2002.

have an associated fairness condition. The ordering annotation and fairness condition will be used to verify liveness properties. See [3, 4] for the formal definition of predicate diagrams.

Verification reactive systems using predicate diagrams is done in two steps. The first step is generating the predicate diagram that represents the abstraction of the system described by specification. The next step is to check whether the predicate diagram satisfies the desired properties or not.

Related to the first step, in [3] Cansell et.al. presented techniques based on abstract interpretation [2] that can be used to compute a predicate diagram for a given system specification and a set of predicates that define the abstraction (rewrite rules). Viewing predicate diagrams as finite-state transition systems, temporal properties of their traces can be established using LTL model checking such as SPIN [9]. The detail of how to encode the diagrams in Promela (the input language of SPIN) and how to verify them using SPIN is explained in [3, 4].

Based on their ideas, I have developed PreDiaG (Predicate Diagram Generator). PreDiaG is a tool prototype that supports the verification reactive systems using predicate diagrams. The two main tasks of PreDiaG are to generate predicate diagrams automatically and to produce the representation of predicate diagram in Promela language. PreDiaG also produces the graphical representation of predicate diagrams.

Related Work. There are some tools based on the similar idea, such as InVest from Saidi et.al. [8, 14] and SAL from Bensalem et.al. [1]. The most related work is the tool from Cansell et.al. [3]. Instead of using the rewriting engine *Logic Solver* from Atelier B [15] and the automatic prover Simplify [7] like their implementation, PreDiaG uses its own simple rewriting engine. Besides the generation algorithm that is implemented in PreDiaG, their tool has also implemented some methods for improving the abstract interpretation; but it does not support the generation of Promela code and the graphical representation of predicate diagrams.

Outline. This paper is structured as follows. Section 2 describes the input and output files that are needed and produced by the tool, the architecture and the algorithm of the generation of predicate diagrams. Some experiment results will be given in section 3 and section 4 concludes this paper.

2 Tool

2.1 Input and output

In order to generate the predicate diagrams, this tool needs three input files, namely: specification file (.tla), state predicate declaration file (.prd) and rewriting rules file (.rew).

The first output of this tool is the representation of predicate diagrams in Promela (.pro) and the second output is the graphically representation of predicate diagrams (.dot).

2.2 Architecture

There are four main components of PreDiaG, namely user interface, abstract states generator, rewriting engine and output generator. Each of these components will be briefly discussed in the sequel.

1. User interface
   This component receives the input files from the user. It gives the information extracted from .tla and .prd to the abstract states generator, whereas the information extracted from .rew file will be given to the rewriting engine component. From the output generator it receives the representations of the generated predicate diagrams as Promela code and as .dot file and displays the both representations on the screen.
2. Abstract states generator
   This component receives the abstract specification and the state predicates declaration from the user interface component. It generates the abstract states and gives the result to the output generator.
3. Rewriting engine
   This component receives a set of rewriting rules from the user interface component and stores them in a table (rule-base). Every rewriting rule consists of two formulas, where the second formula is the simplication of the first formula. During the generation process this module receives formulas from the abstract state generator, simplifies the formulas using the rules in rule-base and give the simplified formulas to the abstract state generator.
4. Output generator
   This component consists of two modules: the module that produces the representation of predicate diagram as .dot file and the module that produces the representation of predicate diagram in Promela language. These two modules then give the produced representations to user interface component.

2.3 Predicate diagrams generation

Let us illustrate the generation process of a simple example, the so called "AnyY problem" [10]. Fig. 1 presents a simple program consisting of two processes communicating by the shared variable x, initially set to 0. Process $P1$ keeps incrementing variable y as long as $x = 0$. Once process $P2$ sets x to 1, process $P2$ terminates and some time later so does P_1 as soon as it observes that $x \neq 0$.

$$
\begin{array}{l}
---------- \ \mathsf{MODULE}\ AnyY \ ----------\\
\mathsf{VARIABLES}\ x, y\\
Init \equiv x = 0 \wedge y = 0\\
P1 \equiv x = 0 \wedge y' = y + 1 \wedge x' = x\\
P2 \equiv x' = 1 \wedge y' = y\\
Spec \equiv Init \wedge \Box[P1 \vee P2]_{\langle x,y\rangle} \wedge \mathsf{WF}_v(P1) \wedge \mathsf{WF}_v(P2)\\
==========
\end{array}
$$

Fig. 2. TLA specification of AnyY problem.

We want to generate a suitable predicate diagram for the AnyY problem. Following the abstraction process in [3], the predicate $y = 0$ might be represented by $y = zero$ and $y > 0$ by $y = pos$. The specification input file (AnyY.tla) appears on Fig. 2, the predicate declaration file (AnyY.prd) appears on Fig. 3 and the rewriting rules file (AnyY.rew) appear on Fig. 4.

```
– – – – – MODULE AnyY – – ––
Init == x = 0 ∧ y = 0
P1 == x = 0 ∧ y' = x + 1 ∧ x' = x
P2 == x' = 1 ∧ y' = y
==========================
```

Fig. 2. The content of file AnyY.tla.

```
(* state predicates *)          (* constraints *)
p1 == x = 0                     p1 <=> ∼ (p2)
p2 == x = 1                     p3 <=> ∼ (p4)
p3 == y = zero
p4 == y = pos
```

Fig.3. The content of file .AnyY.prd.

```
(* abstraction function *)      (* rewrite rules *)
y = 0 => y = zero               0 = 0 => true     zero + 1 => pos
∼ (y = 0) => y = pos            1 = 0 => false    pos + 1 => pos
```

Fig. 4. The content of file AnyY.rew.

Abstract states are represented as a record of n character, where n is the number of state predicates which are declared in .prd file. Every i^{th} character of an abstract state s corresponds with the i^{th} state predicate. If s[i] = '1' then the correspondence predicate holds, otherwise if s[i] = '0' then the correspondence predicate does not holds on that state.

The first step in generating predicate diagram is to find a set of initial states, which are abstract states that satisfy Init and constraints that are specified in .prd. In the case of program AnyY, the only initial state is 1010.

Starting from the set of initial states, we repeatedly apply this following steps:

1. Add the value of current abstract state to the rule-base. For example, if the current state is 1010 then the rule-base contains x => 0 and y => zero.
2. For every action formula A (except Init) we evaluate every sub-formula that contains only unprimed variables using the rules in the rule-base. If the result is true then we continue with the evaluation of every sub-formula of A that contains some primed variables. Based on the unprimed version of

the resulted formula, we find a set of abstract states that satisfies it and also satisfy the predicate constrains and then add those states to the set of abstract states.
Assume the current state is 1010 and the current action formula is P1. The evaluation of unprimed part of P1 is as follows:

$$x = 0 \longrightarrow 0 = 0 \longrightarrow \mathsf{true}$$

and the evaluation of primed part of P1 is as follows:

$$y' = y + 1 \longrightarrow y' = \mathsf{zero} + 1 \longrightarrow y' = \mathsf{pos}$$
$$x' = x \longrightarrow x' = 0.$$

The only abstract state that satisfies the formula $y = \mathsf{pos} \wedge x = 0$ and the predicate constraints is 1001.

The generated predicate diagrams for the AnyY problem is given in Fig. 5.

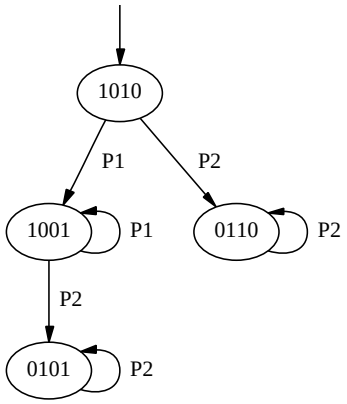

Fig. 5. The generated predicate diagram for AnyY problem.

3 Results

Some experiments have been done with PreDiaG. Using this PreDiaG and SPIN, I have verified the safety and liveness properties of some protocols, such as "Dining Mathematicians" [6], "Bakery" [11] and "Peterson" [13].

4 Conclusions and Future Work

I have presented PreDiaG which is a tool prototype for the generation of predicate diagrams. Using this tool, predicate diagrams can be generated automatically. One of the outputs of this tool is the representation of the generated diagrams in Promela language. The verification is then can be done by using the model checker SPIN.

Now I am studying the applications of predicate diagrams on parameterized systems and plan to extend PreDiaG so that it can support the generation of predicate diagrams for parameterized systems.

References

1. S. Bensalem, et.al. An overview of SAL. In C M. Holloway, editor, *LFM 2000: 5^{th} NASA Langley Formal Methods Workshop*, pages 187-196, 2000.
2. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symp. Princ. of Prog. Lang.*, pp. 238-252. ACM Press, 1977.
3. Dominique Cansell, Dominique Méry and Stephan Merz. Predicate diagrams for the verification of reactive systems. In 2^{nd} *Intl. Conf. on Integrated Formal Methods (IFM 2000)*, vol. 1945 of *Lectures Notes of Computer Science*, Dagstuhl, Germany, November 2000. Springer-Verlag.
4. Dominique Cansell, Dominique Méry and Stephan Merz. Diagram refinements for the design of reactive systems. In *Journal of Universal Computer Science*, 7(2):159-174, 2001.
5. Michael A. Colón and Tomás Uribe. Generating Finite-State Abstractions of Reactive Systems using Decision Procedure. In *10th International Conference on Computer Aided Verification*, vol. 1427 of Lecture Notes in Computer-Science, pp. 293-304, Springer-Verlag, June/July 1998.
6. D.R. Dams, O. Grumberg and R. Gerth. Abstract interpretation of reactive systems: Abstractions preserving $\forall$CTL*. In *IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET 94)*, pp. 573-592, June 1994. Park. Experience with Predicate Abstraction. In *11th International Conference on Computer Aided Verification*, vol. 1633 of Lecture Notes in Computer-Science, pp. 160-171, Springer-Verlag, 1999.
7. D. Detlefts, G. Nelson, and J. Saxe. Simplify: the ECS theorem prover. Technical report, Systems Research Center, Digital Equipment Corporation, Palo Alto, CA, November 1996.
8. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Conference on Computer Aided Verifications*, vol. 1254 of Lecture Notes in Computer-Science, pp. 72-83. Springer-Verlag, 1997. June 1997, Haifa, Israel.
9. G. Holzmann. The SPIN model checker. *IEEE Trans. on software engineering*, 16(5):1512-1542. May 1997.
10. Y. Kesten and A. Pnueli. Taming the Infinite: Verification of Infinite-State Reactive Systems by Finitary Means. In *Engineering Theories of Software Construction, (NATO) Science Series*, Series III: Computer and Systems Sciences, Vol. 180, pages 261-299, IOS Press 2001.
11. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):435-455, May 1974.
12. L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3) : 872-923, May 1994.
13. Gary L. Peterson. A new solution to Lamport's concurrent programming problem using small shared variables. *ACM Transaction of Programming Languages and Systems*, 5(1):56-65, 1983.
14. H. Saidi and N. Shankar. Abstract and model check while you prove. In N. Halbwachs and D. Peled, editors, *Conference on Computer-Aided Verification (CAV'99)*, vol. 1633 of Lecture Notes in Computer-Science, pages 443-454, Trento, Italy, 1999, Springer-Verlag.
15. STERIA - Technolgies de l'Information, Aix-en-Provence (F). *Atelier B, Manual Utilisateur*, 1998. Version 3.5.