Extracting a Petri Net Representation of Java Concurrency: Literature Survey

Anita J. Bateman
abateman@cs.utexas.edu

Travis Pouarz
pouarz@ece.utexas.edu

25 March 2002

Abstract

Automated analysis is necessary to verify the correct operation of concurrent systems. Petri Nets have a large body of research, both theoretical and practical, supporting their use for concurrent system analysis. The Java language provides built-in capabilities for developing concurrent systems with threads. Researchers have examined how to extract Transition Systems from Java source and how to construct Petri Nets from Transition Systems. This paper examines that research and then proposes designing an automated system to extract a Petri Net from Java source by building upon that prior work.

Introduction

The complexity of concurrent systems creates uncertainty over how to verify system correctness or perform analysis of a particular system property. Analysis of concurrent systems frequently suffers from *state explosion*, which requires exponential resources to solve [1]. Petri Net (PN) research has yielded valid and accurate models of concurrent systems and has provided various analysis techniques and tools. Researchers have begun to examine the Java language, which has built-in support for creating multi-threaded software. Their research has produced algorithms to extract Transition Systems (TS) from Java source [2] and to construct Petri Nets (PN) from Transition Systems [3].

In this paper, we will discuss Transition Systems and Petri Nets as models, identify the subset of the Java language we are focusing on, expand on the algorithms used to extract a TS from Java source and constructing a PN from a TS, briefly discuss concurrency analysis with Petri Nets and propose a design for an automated system to extract a PN from Java source, building on prior research.

1 Models

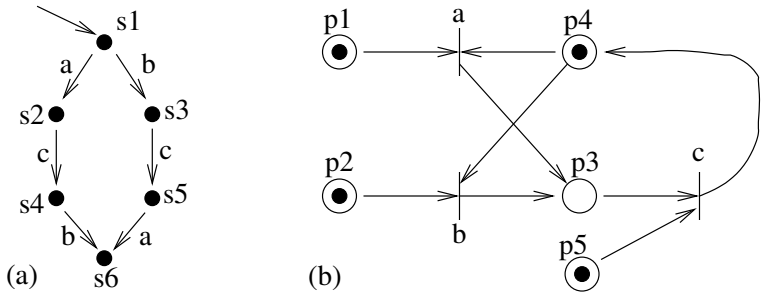

Figure 1: From [3] (a) Transition System (b) Petri Net

We are concerned with two types of models: Transitions Systems and Petri Nets. Examples of each are presented in Figure 1.

A Transition System is a tuple $TS = (S, E, T, s_{in})$ of a set of *states* and a set of *events*, a $T \subseteq S \times E \times S$ *transition* relation, and an *initial state*. For our purposes, we will not allow any self-loops or multiple arcs between two states [3]. A Finite State Machine (FSM) is a well-known type of TS where the set of states must be of finite size.

A Petri Net is a tuple $N = (P, T, F, m_0)$ of a finite set of *places*, and a finite set of *transitions*, an $F \subseteq (P \times T) \cup (T \times P)$ flow relation, and an *initial marking*. It is a graph model with two types of nodes, places (circles) and transitions (blocks or lines). Directed edges are allowed between place and transition nodes, but not between nodes of the same type. A place containing a token traditionally contains a solid dot. A marking is the set of places which contain tokens [3].

Petri Nets are simulated according to the following firing rule: a transition is *enabled* when all incoming places contain at least one token. Upon firing the transition, a token is removed from all incoming places and a token is added to all outgoing places.

Petri Nets are strictly more powerful than FSMs. A *safe* Petri Net has exactly the same expressive power as an FSM. A Petri Net is *safe* if for each reachable marking, each place contains no more than one token [1].

Using an event-based PN model rather than a state-based TS model makes explicit the relationship between events, is often more compact , and has some properties that can be verified structurally—without expensive enumeration of all states [3].

2 Java Concurrency

We assume the reader has a basic knowledge of the Java programming language constructs and concurrency APIs (the Thread class and Runnable interface). Since Java's concurrency capability is rather broad, we define a subset of Java on which to focus. Corbett [2] defined a subset of Java concurrency features that is centered around two common concurrency design patterns: the producer/consumer pattern and the observer pattern. The first pattern uses mutual exclusion to control access to shared variables and the second pattern highlights thread communication with `wait()`/`notify()`/`notifyAll()` calls. We adopt the same assumptions as [2] for the subset of Java that we will focus on.

We also adopt the scheduling assumption that threads are scheduled arbitrarily by the system thread scheduler and that there are no modifications of thread priority settings. Additional thread methods are excluded because they can cause added complexity in the concurrent model of the system. These methods include `join()`, `yield()`, `suspend()`, `resume()`, and `stop()`.

With multiple threads of execution, synchronization between threads that use the same resources is a common problem. Java handles these issues through locks and the `synchronized` keyword. Each object in Java has a lock, which can be obtained by any Java thread. The `synchronized` keyword allows only one thread to enter a `synchronized` block of code at a time.

3 Transition Systems from Java

Corbett [2] demonstrated that an FSM can be constructed from a Java source program by using *shape analysis*. Shape analysis attempts to preserve the shape of common linked

structures in a program. Corbett's work created a model builder, not a model extractor. This forces the engineer to extract the relevant parts of the Java program and create the necessary abstractions by hand before feeding the Java source into the model builder [2]. The model builder will accept the Java source and a property to verify, e.g. deadlock, and will produce a Transition System model.

Initially, the abstract syntax tree (AST) of the program is constructed. A traversal of this AST constructs a control flow graph (CFG) for the `main()` method and for each thread's `run()` method. Method calls and constructors are inlined as they are encountered, which may produce an exponential increase in the number of statements. Each arc in the CFG is a transformation that represents a bytecode instruction. Transformations may be marked *invisible* if they will not change the meaning of the CFG with relation to the property being verified. Invisible transformations are collapsed explicitly into a single visible transformation, reducing the size of the state space and CFG. A depth-first search of the program's state space will generate the transitions for the TS model. At each state, a transition is generated for each ready thread that represents the execution of that thread up to the next visible transformation [2].

As an option when applying the model building approach just discussed, Corbett also defined several state space reduction techniques in order to reduce the size of the model constructed. These techniques include owned-variable reduction, protected-variable reduction, relock reduction, notify reduction and unlock reduction. The reductions are applied to the CFGs that represent the program and approximate storage structure graphs (SSG) are constructed to track all possible object reference paths through the heap at a particular statement [2]. Shape analysis has been shown to have a worst-case running time of $O(S^2V^4)$ and a worst-case space requirement of $O(SV^2)$, where S is the number of statements after in-

lining all procedure calls and V is the number of variables and allocators. The reductions for determining which transformations access owned versus protected variables have worst-case running times of $O(SV^2)$ and $O(SV)$ respectively [2].

Corbett claims that despite the complexity of the algorithm as indicated, the average cost of shape analysis is acceptable because SSGs are generally sparse, S and V refer to the number of modeled statements (vs. all statements) and the overall complexity will be dominated by the model building/checking step rather than the reduction of states [2]. This research has also shown that all Transition Systems derived using the reductions described preserve the accuracy of the model with respect to the property being examined.

4 Petri Nets from Transition Systems

Cortadella *et al.* have developed a method for synthesizing a Petri Net from a number of Transition System models [3]. Their method builds upon the theory of regions which was first described by Nielsen *et al.* in a paper about Elementary Transition Systems (ETS) [4]. Given a subset of states, a TS event can be characterized by whether it always enters, always exits, is always internal, or is always external to that subset. A *region* is a subset of states such that all transitions labeled with the same event have the same *enter*, *exit*, *internal*, or *external* label with respect to that region. States may be split to make the labels work out conveniently. The regions will become transition relations in a Petri Net [3]. An important property of regions refined by Bernardinello *et al.* [5] is that

Property 1 *every region can be represent as a union of disjoint minimal regions [3].*

Synthesis of an *elementary net* from an ETS has been explored [4] and extending those results to Petri Nets in particular is straightforward [3]. The extended algorithm generates

a *saturated* net where all regions are mapping to corresponding places. The net will have as many places as regions. The events that *enter* and *exit* regions will become transitions. An event that *enters* a region will become a transition with an arc to a place. An event that *exits* a region will become a transition with an arc from a place to the transition.

Since all regions are mapped to places, the PN is *saturated*. Any ETS has a unique saturated net. However, such a net is full of redundancy. Further, since all regions of a TS must be considered and elementary checks must be performed on every state, the algorithm has such a poor efficiency that it is impractical [3].

Using the the minimal region property (Property 1), the algorithm can be improved to create a *minimal saturated* net. Cortadella *et al.* extended their result to create *place-irredundant* and *place-minimal* Petri Nets [3].

Further, they found that they could extend their method so that not only can it derive a PN from an ETS, but it can derive a PN from any TS that has a bisimilar ETS. They called this class of transitions systems *excitation-closed* transition systems (ECTS). The breakthrough in allowing and using ECTSs is that the required conditions can be efficiently checked. Rather than being required to check for elementarity on every state in an ETS, they can check for excitation-closure on every event. In concurrent systems, the number of events is often much fewer than the number of states [3].

Figure 2 shows the framework they present for their implementation of a software system that derives a PN from a TS.

The goals of the system were to achieve "good" results in reasonable time. A good result is presumably useful for analysis but not necessarily optimal. Some compromises were made to achieve this result. TS label splitting was done by considering regions that are predecessors of an event (pre-regions). Merging minimal pre-regions is done with a greedy

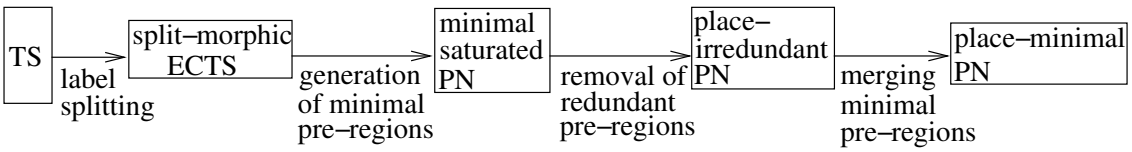

Figure 2: From [3], a framework for creating a PN from a TS

strategy that will not always yield an optimum their transformations to ensure that deadlock and liveness properties are preserved in the extracted PN [3].

The system can be tuned depending on the desired use of the resultant PN. It can produce PNs that are either somewhat-redundant in form but more comprehensible to human users or maximally compact for efficient computer manipulation.

5 Concurrency Analysis with Petri Nets

Pezzè [1] has indicated that it "is always possible to represent an Ada program composed of a fixed number of tasks by means of a safe [Petri] net." We assume that this can be extended to Java through the similarity in concurrency constructs between Ada and Java. One area of Petri Net usage in concurrent system analysis is invariant analysis. This kind of analysis represents a net as a set of equations and attempts to find integer solutions to the equations.

Invariant analysis can be used to analyze race conditions and deadlock. The analysis is still exponential in the worst case and is also pessimistic for detecting deadlock. However, the solutions may help to guide and prune a reachability analysis of the system [1]. Reachability analysis can provide stronger assurance than invariant analysis for critical properties of concurrent systems, although it can still be pessimistic by sometimes failing to accept a correct program. The ability to execute both invariant analysis and reachability analysis on the same Petri Net model of a concurrent system is a great advantage [1].

Deadlock detection, race detection, and other concurrency problems are known to be theoretically NP-complete. This is often called the "state explosion problem" of reachability analysis [1]. No implementation or model will be able to improve on the worst-case exponential resources (time or memory) required to solve these problems. However, many practical problems have been shown to be solvable with reasonable resource usage. It is in the solution of these practical analysis problems that exploitation of various implementations and models, such as Petri Nets, show useful performance and usability differences [3].

6 Conclusion and Future Plans

Corbett and a group at the University of Kansas have implemented an open-source software system for the generation of a TS model from Java source code using the methods discussed in Section 3. They call it *Bandera*. Cortadella and his fellow researchers have implemented a software system called *Petrify* that is able to derive a Petri Net from a Transition system, as discussed in Section 4. Source for *Petrify* is not publicly available. Interestingly, we see no cross-references or citations between these groups of researchers.

Our aim is to link their systems together. We plan to write a new backend module for the *Bandera* system that will write a transition system suitable for filtration through the *Petrify* binary. The result will be a Petri Net suitable for further analysis by hand or by another automated system that might, say, perform a reachability analysis for deadlock detection.

We will implement a small classic concurrent problem in Java, both correctly and incorrectly. We will then examine what information can be gleaned by hand inspection of the Petri Net which results from our end-to-end process.

References

[1] M. Pezzè, R. N. Taylor, and M. Young, "Graph Models for Reachability Analysis of Concurrent Programs," *ACM Transactions on Software Engineering and Methodology*, vol. 4, no. 2, pp. 171–213, Apr. 1995.

[2] J.C. Corbett, "Using Shape Analysis to Reduce Finite-state Models of Concurrent Java Programs," *ACM Transactions on Software Engineering and Methodology*, vol. 9, no. 1, pp. 51–93, Jan. 2000.

[3] J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev, "Deriving Petri Nets from Finite Transition Systems," *IEEE Transactions on Computers*, vol. 47. no. 8, pp. 859–882, Aug. 1998.

[4] M. Nielsen, G. Rozenberg, and P. S. Thiagarajan, "Elementary Transition Systems," *Theoretical Computer Science*, vol. 96, no. 1, pp. 3–33, 1992.

[5] L. Bernardinello, G. De Michelis, K. Petruni, and S. Bigna, "On Synchronic Structure of Transition Systems," *Proc. Int'l Workshop Structures in Concurrent Theory*, pp. 69–84, May 1995.