# Extracting a Petri Net Representation of Java Concurrency

Anita J. Bateman abateman@cs.utexas.edu

Travis Pouarz pouarz@ece.utexas.edu

08 May 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 describes a way to link these research efforts using a newly developed method of efficiently enumerating all reachable state transitions. It uses a novel method of state encoding with Binary Decision Diagrams (BDDs) to achieve its result.

# Introduction

The complexity of concurrent systems creates uncertainty over how to verify system correctness or perform analysis of particular system properties. Petri net research has yielded models of concurrent systems which are useful for analysis, either visually or with tools [1].

The Java programming language has built-in support for creating concurrent, multi-threaded programs. Current research has produced methods and systems to extract transition systems (TS) from Java source [2]. Other recent research details how to construct Petri nets from transition systems [1]. While both research areas speak of transition systems, they are not directly compatible with each other. We have developed a method to take a complex Von Neumann transition system and produce a state graph TS. We have prototyped the complete system and tested it on simple concurrent problems such as deadlock and producer/consumer.

# 1 From Java to Petri Net

The system that we have constructed for creating a Petri net representation of a Java program involves three programs and four data representations, as shown in Figure 1.

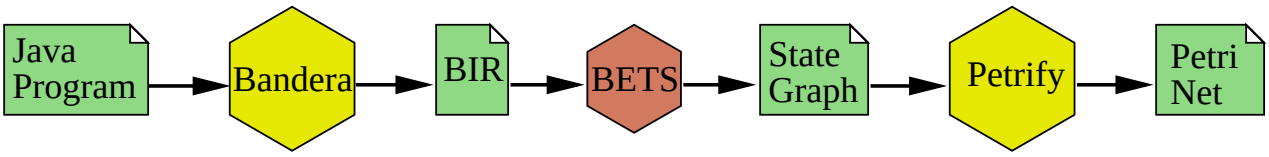


Figure 1: Create a Petri net from a Java program with three programs and two intermediate data representations.

Bandera, a research program being developed by Corbett [2] and a group at the University of Kansas, transforms a Java program into a Von Neumann Architecture-like transition

system called Bandera Intermediate Representation (BIR). Bandera performs state space abstractions to eliminate the parts of the Java program that are not directly of interest, such as those operations on local variables which have no effect on the concurrent operation of the program [2].

Petrify, a program developed by Cortadella [1], takes a state graph transition system and produces a Petri net. Petrify has a number of options to control the properties of the resulting Petri net. In general, we direct Petrify to generate "free choice" Petri nets because they are the easiest form to digest visually and trace by hand.

We have developed the component in the middle portion of the diagram, called BETS (the BIR Extraction to Transition System). BETS extracts a Petrify-compatible state graph from BIR, so that a concurrent Java program may be modeled as a Petri net.

# 2 BIR and State Graph Models

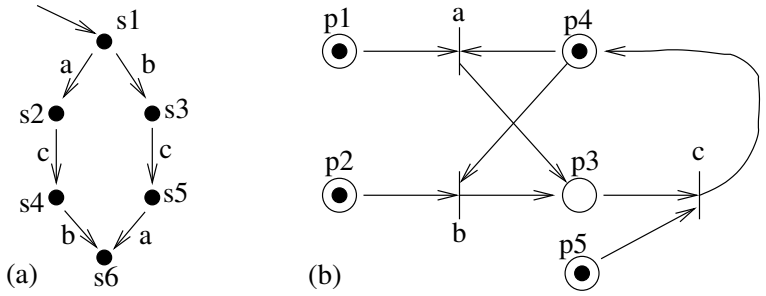


Figure 2: From [1]: (a) State graph transition system and corresponding (b) Petri net.

Mathematically, a *transition system* is a tuple $TS = (S, E, T, s_{in})$ of a set of *states* $S$ and a set of *events* $E$, a $T \subseteq S \times E \times S$ *transition* relation, and an *initial state* $s_{in}$. You can see a graphical depiction of a TS in Figure 2(a). This simple TS form can be called a *state graph*.

The Bandera system generates a more complex TS to represent a Java program, called

Bandera Intermediate Representation (BIR). It looks more like a Von Neumann Architecture program than a state graph:

```
Process Deadlock()
buffer : lock A;
buffer : lock B;

main thread T0
loc0: when lockAvailable(A) do { lock(A); } goto loc1;
loc1: when lockAvailable(B) do { lock(B); } goto loc2;
loc2: { unlock(B); unlock(A); } goto loc0;
end T0;

thread T1
loc3: when lockAvailable(B) do { lock(B); } goto loc4;
loc4: when lockAvailable(A) do { lock(A); } goto loc5;
loc5: { unlock(A); unlock(B); } goto loc3;
end T1;
end Deadlock;
```

We treat the locations, `loc`$X$ statements, as *program counter* (PC) "addresses." Each location has one or more associated actions, each with an optional guard. At each location/address, an action is available, possibly behind a guard. Running threads execute the actions at a specific location atomically. The interleaving of the actions is intentionally arbitrary and non-deterministic.

The task for BETS is to transform the BIR into a state graph like that shown in Figure 2(a). BETS produces a text listing of all state transitions in this tuple format:

`<originating state name> <transition name> <destination state name>`.

For example, a tuple might look like this: `s0 P1_loc1 s1`. There are some minor shortcuts available in the format, but the ultimate size of the output is on the order of the number of transitions. So, if the number of states or the number of transitions is of an unmanageable order of magnitude, then no matter what BETS does, the problem will not be practically solvable. For problems that are practically solvable, BETS aims to perform the transformation as efficiently as possible. The resulting state graph can then be fed into Petrify to

generate a Petri net, like that shown in Figure 2(b).

## 2.1 State Representation

BETS represents states by boolean functions in a new and novel way. A meaning of a state function is most apparent when it is expressed as a sum-of-cubes formula. The boolean variables are represented by simple 1-variable cubes which *are* or *are not* present. State components that have a value are represented by *tagged* cubes. There is a special variable, the tag, which represents the purpose of the cube. The rest of the variables in the cube encode the value associated with the tag. The current implementation supports 32-bit values, but that is an arbitrary limit than can be changed, even dynamically.

Values are encoded by a set of variables which represent bits in the value. A representative "power" variable,$p_x$, is only present in the cube if the $x$th bit is a 1. The decimal value 10 can then be represented by the cube $p_3 p_1$.

Of particular interest is that state functions contain a value which represents the PC for each running thread. The PC-space, program address space, can be "shared" so that multiple copies of the same thread can be run concurrently. The Bandera code cannot support multiple copies of a thread at the moment, but if Bandera is improved, BETS can accommodate.

Here is an example of a state function:

$$s_0 = t_0 p_2 + t_1 p_1 + l_a$$

The state $s_0$ has two running threads. Thread $t_0$ is at PC 4. Thread $t_1$ is at PC 2. The lock $l_a$ is held by the thread whose PC requires it to be holding the lock in a protected critical section. There is no reason to encode which thread owns the lock in the model that we used. Certainly, other modeling goals may use variations of this particular encoding scheme.

All functions are organized and represented by Binary Decision Diagram (BDD) data structures. The BDDs are actually Reduced-Ordered BDDs managed by a BDD manipulation library, organized as a "BDD package" [3]. Numerous BDD packages are available and they all make available a relatively common API. A function represented by a BDD takes the form of a handle to a BDD node. BDDs have proven a very effective and efficient way to represent boolean functions in many areas, including symbolic simulation, logic optimization, combinational logic verification [4], and graph transformation (Petrify) [1].

For example, to extract the value cube (cube of only $p$ power variables) of a tagged cube in a sum-of-cubes, BETS does this:

$$\text{BDD pcube} = \text{exists}(\text{restrict}(!\text{cofactor}(\text{sumOfCubes}, !\text{tagVariable})), \text{tagVariable})$$

Existential, restriction, and cofactoring operations are normal elements of BDD package function manipulation.

## 2.2 State Exploration

The reachable states of the graph are explored in a manner similar to a breath-first search. The initial state is added to the set of to-do states. This state is probably a state in which a special, *main* thread has a PC of zero and is thus ready to run its first instruction.

While the to-do set is not empty, an arbitrary state is selected from that set for processing. Once a state is selected from the to-do set, it is added to the previously-done set so that requests to add the state to the to-do set again can be ignored.

As described previously, a state is a function represented by a BDD node. The BDD package APIs give out handles to nodes, often in the form of a pointer. The API that BETS uses hands out opaque 32-bit unsigned values as BDD node handles. BETS encodes the 32-bit BDD node values as a BDD cube. It uses the $p$ power variables mentioned earlier,

but all 32 power variables are present in every cube as either positive or negative literals. This allows the selection routine to work, though we will not be covering the details of that selection in this paper.

The sum of these value cubes is a function representing a set of states. Using a sum-of-cubes where each cube represents a state and the sum represents a set of states is standard practice in the world of symbolic model checking. In that world, each cube represents a state directly. In our world, each cube represents a handle to another BDD-represented function where that function represents our state. As far as we know, this two-layer set-of-states encoding is novel.

To process a state, each running thread is given a chance to "run." If a guard at the thread's PC is met (there may be more than one), then the action associated with the PC is "executed." The execution of an action creates a new state by modifying a copy of the existing state. The new state is placed in the to-do set if it is not already in the previously-done set. With our BDD-function implementation of state, a "copy" of a state is simply another reference to the same BDD: the copying has almost zero cost. The transition from one state to another is recorded in the output with a characteristic name for the action that took place. For example, say that two actions are possible at PC 0x4a with mutually exclusive guards. When executed by thread 0, the name of the first state-transforming action will be called "t0_4a_1" and the other action will be called "t0_4a_2". The action name is what will appear in the Petri net as a transition.

The granularity of the action should depend on the aspect of the system that is being tested. Every action is executed as an atomic step. To test that the actions of a critical section are not inappropriately interrupted by another thread, the critical section may be broken up into a series of actions with different PC addresses. On the other hand, to ensure that the various critical sections are executed such that the program functions correctly and

does not deadlock, the critical sections may each exist as a single action in the modeled program. The Bandera model will tend to break up the actions into the former, more finely grained model, but it may be worthwhile to devise a mechanism to be able to get a more coarsely grained model.

## 2.3 Small Example: Deadlock

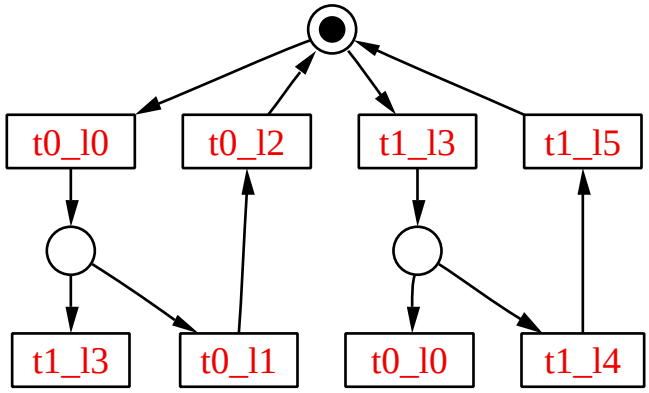


Figure 3: Petri net for the Deadlock BIR example in Section 2

Figure 3 shows result of the process on the Deadlock example presented as BIR earlier. The free-choice Petri net in the figure shows three places where the thread scheduler has the option of nondeterministically choosing how the program progresses. It also shows that in two of the places, the scheduler can make a choice that leads to a dead-end. In this case, the dead-end is a deadlock situation. In other cases, it may signify the successful completion of a program that is not intended to run forever, or it may mean that the BETS has stopped exploring state artificially because a particular preset resource limit was hit. This technique can be used to help alleviate state explosion. For example, if a counter could theoretically count up to $2^{32}$, Bandera and BETS allow annotations to restrict the counter range to something smaller, such as $2^4$.

### 2.4 Efficiency

It is hard to say at what point a BDD representation of a function will become inefficient. There are a few functions which are known to make BDDs "blow up" in size, such as integer multiplication [5]. BDD size is determined by the specific function represented and small perturbations can make a large difference in the efficiency of the representation. Practice has shown that BDDs do a good job for many practical problems [3][4]. Heuristically, keeping the number of additional thread variables as small as possible is considered a "good thing" because the worst case size is exponential on the number of variables. Our encoded representation uses a small set of variables. In particular, adding threads which have no local state in the modeled system cost only one additional "tag" variable per thread.

## 3 Conclusion and Future Work

We implemented a trial version of the methods described here to discover their workability. Far from revealing difficulties, the implementation exercise revealed our design's flexibility and led to a series of improvements. The particular usage of BDDs for state encoding and state exploration is a new novel technique that appears well-suited for the task described here.

For this experiment, we implemented code for BETS either by hand from a textual BIR output. This could be automated by integrating the BETS subsystem with the Bandera framework as the interface between Bandera and Petrify. It should just be a "simple matter of programming," however the Bandera framework is complex enough that the task would probably be somewhat labor-intensive. Further, given that Bandera is Java-based but the well-developed, well-known BDD packages (such as CUDD [6]) are all C/C++ native code, a hybrid system is probably most prudent.

# References


[1] 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.

[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] R. E. Bryant, "Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams," *ACM Computing Surveys*, vol. 24, no. 3, pp. 293–318, 1992.

[4] K. S. Brace, R. L. Rudell, R. E. Bryant, "Efficient Implementation of a BDD package", *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pp. 40–45, 1990.

[5] R. E. Bryant, "On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication," *IEEE Transactions on Computers*, vol. 40, no. 2, pp. 205–213, Feb. 1991

[6] F. Somenzi, *CUDD: CU Decision Diagram Package, Release 2.3.1*, Univ. of Colorado at Boulder, `http://vlsi.colorado.edu/~fabio/CUDD/`, 2001.