Towards Software Model Checking using MDGs

M. Krykhtin[1], Y. Mokhtari[1], O. Aït Mohamed[1] and X. song[2]
[1] ECE Dept., Concordia University, Canada, {m_krykht, mokhtari,ait}@ece.concordia.ca,
[2] ECE Dept., Portland State University, USA, song@ece.pdx.edu

Abstract— **In this paper, we discuss the integration of Multiway Decision Diagrams (MDG) model-checker into Bandera framework. A schema is introduced for transforming the Bandera Intermediate Representation (BIR) into the language of MDG model checker. Experience with model-checking the Java programs demonstrates that this approach offers effective support for verifying software models.**

I. Introduction

Model checking is an automatic finite-state verification technique that has gained wide acceptance as a powerful tool for debugging hardware design [1]. It consists of exhaustively checking a finite-state model of the system for violation of its requirements specified in some temporal logics. This approach provides a level confidence comparable to that of machine-checked proof of the system's requirements correctness without (extensive) human guidance.

During the past few years, there has been increasing interest in the application of model checking to software. However, the transfer of this technique to software was very slow due to the state explosion problem and the semantics gap between the artifacts produced by software developers and those accepted by current verification tools [3]. These problems have been tackled in several ways. These attempts can be roughly classified into two approaches: the first approach [2] consists of developing new dedicated techniques for software while the second one [3], [4] allows the integration of existing tools by using high (abstract) modeling techniques. The advantage of the former is that the difficulty of applying software model checking is addressed directly, while the advantage of the latter is reusing an existing tool and therefore the effort is oriented towards the integration part.

In this paper, we describe an experience with the second approach using the toolset Bandera [3] and Multiway Decision Diagrams (MDGs). The rest of paper is organized as follows. Section II surveys MDG and Bandera tools and emphasizes the Java concurrency model. Section III describes the translation schema with a focus on Java concurrency model. Section IV considers a simple Java program and illustrates its modeling and verification using MDG tool. Finally, Section V concludes the paper and states the future work.

II. Background

A. Java Concurrency Model

Java is a general-purpose, concurrent, object-oriented language. The key feature of concurrency is *threads*. The Java class *java.lang.Thread* is used to initiate and control new activities. A thread is created by executing a *new Thread()* allocation statement and it is started by invoking the thread object's *start()* method which begins execution of the *run()* method. To synchronize threads, Java uses *monitors*, which are a high-level mechanism for enforcing mutually exclusive access to a region of code. The behavior of monitors is explained in terms of *locks*. Each Java object has a unique lock associated with it. In addition, each object has an associated *block-set* and *wait-set* for managing threads that are blocked on the object's lock or waiting on the object's lock. When a *synchronized(expr)* statement is executed by a current thread, the object expression is evaluated and the resulting object's lock is checked for availability. If the lock has been acquired by some other threads, the current thread becomes blocked and it is inserted into the object's block-set. When a thread leaves the synchronized block, it unlocks the lock associated with it. Acquiring the lock of the current object during a method body may be abbreviated by placing the keyword *synchronized* in the method's signature.

A thread may become waiting on object's lock by invoking *wait()* method, which results in putting the thread in the object's wait-set and releasing the lock. A waiting thread is released by another thread by invoking *notify()* or *notifyAll()* methods. Invoking *notify()* removes an arbitrary thread while *notifyAll()* removes all the threads from the object's wait-set.

B. MDG-HDL Language

MDGs [5] are a canonical representation of a certain class of many-sorted first-order logic formulas, where data values and operations are represented by abstract variables and uninterpreted functions, respectively. In MDG-based verification, abstract description of states machines (ASM) are used for modeling systems. MDGs have been investigated from different angles and it culminated in a MDG tool providing Prolog-style MDG-HDL for modeling and different verification techniques including sequential and combinational equivalence checking, invariant checking and a subset of first-order LTL model checking.

MDG-HDL specification describes a concrete model by using structural and/or behavioral description. In the structural description, a model is described at the RT level as a collection of components interconnected by nets that carry signals. Each signal is represented by a variable. Variables denoting control signals have concrete sorts, while variables denoting data values have abstract sorts. The control operations are represented by uninterpreted cross-operators while data operations can be viewed as black-boxes and thus represented as uninterpreted functions. Furthermore, MDG-HDL language provides prede-

0-7803-8322-2/04/$20.00 ©2004 IEEE.

fined components (e.g. registers, gates, multiplexers), which are automatically transformed into their MDG representation. Figure 1(a) shows a comparator that produces a control signal z from two data inputs x and y. Both x and y are variables of abstract sort while z is a Boolean variable. An uninterpreted cross-operator *eq* is used to denote the functionality of the comparator. If the meaning of *eq* matters, rewrite rules, such as $eq(x,x) \rightarrow 1$ should be used. An MDG of the comparator is shown in Figure 1(b).

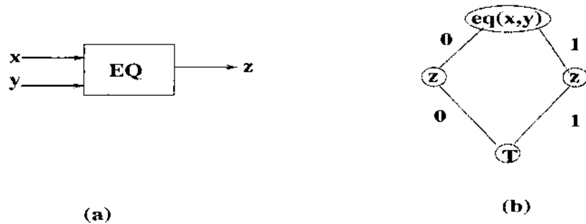

Fig. 1. The MDG for a comparator.

A behavioral description is given by high-level constructs as ITE (If-Then-Else) formulas, CASE formulas or tabular representations. The tabular construct is similar to a truth table but allows first-order terms in rows.

C. The choice of Bandera

We aim to use MDG tool for software model checking. We have decided to use Bandera because like MDG specification it is based on transition system. Moreover, Bandera [3] is a collection of program analysis and transformation components that enables the automatic extraction of finite-state models of Java programs and generates a program model in the input of language of one of existing verification tools, mainly SPIN and SMV. The key component is that the extracted model is specified in Bandera Intermediate Representation and thus enabling different model checker as a back-end. Therefore, we describe the transformation of a significant subset of BIR to the input language of MDG tool which is used to check the resulting model against its requirements (see Figure 2).

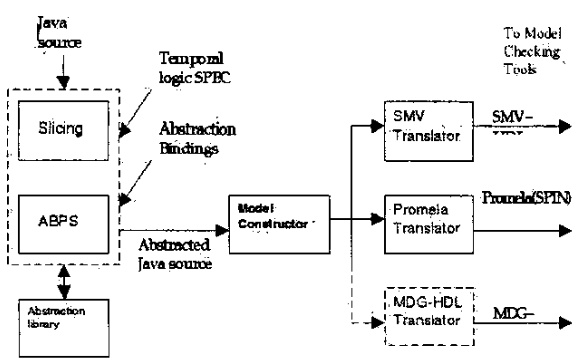

Fig. 2. Bandera Architecture ToolSet

Bandera Intermediate Representation (BIR) is guarded command language intended to facilitate translation of Java programs to different verifiers. BIR uses an asynchronous model and contain constructs convenient for representing Java (e.g. threads, locks, references). A BIR specification describes a *transition system*. Formally, a transition system is a pair (S,T) where :

- $S = D_1 \times \ldots \times D_n$ is set of states. A state is an assignment of values to a finite set of state variables $v_1, \ldots, v_n$ where each v_i ranges over a finite domain D_i
- $T \subseteq S \times S$ is a transition relation. T is defined by a set of guarded transformations $t_i : g_i \Rightarrow h_i$ of the state variables, where $g_i : S \rightarrow Bool$, called the guard, is a boolean predicate on states, and $h_i : S \rightarrow S$, called the transformation, is a map from states to states such that: $T(s,s')$ iff $\exists i. g_i(s) \wedge s' = h_i(s)$

In practice, a BIR transition system is described by a set of objects (the passive part) and a set of threads (the active part). Threads interact only through objects except for locks which are used to affect the state of the other threads (see section II-A).

III. Translation Schema

A. Process

A process represents a single Java virtual machine and consists of a name, a list of definitions, a list of threads, and (optionally) a set of predicates. A definition specifies either an integer constant or type specification (e.g., enumerated type, range, locks, references). An object is specified by a name, a type and (optionally) an initial value. Typically, a BIR program has the following form:

```
process Example()
  SIZE : 2;
  java_lang_Object_ref =
    ref { Lock_col, Lock_col_0, Process1_col, Process2_col };
  state : range -1..5 := 0;
  ....
  main thread Deadlock()
    ...
    loc s25: live  E_Bool
      when (E_Bool == 1)
        do  println();  goto s26;
    ...
    loc s44: live
    loc s66: live  p2
      when true do    goto s30;
  end Deadlock;
  ...
  thread Process1(this : Process1_ref)
  ...
  end Process1;
  ...
end Example
```

A process is converted into set of files that describe a MDG-HDL model, namely, an algebraic specification file, a symbol order file, a circuit description file and an invariant specification file. This conversion may require human guidance in particular for defining variables order.

B. Definitions and Objects

For a constant of a concrete sort, a translator adds a constant_signal component declaration into

a the circuit description file. In the example above, the statement (SIZE=2) is translated to: $component(c_s, constant_signal(value(2), signal(SIZE)))$. The translation of objects is achieved according to their types and will be described in the following.

C. Types

BIR allows seven basic types of objects: boolean, range, enumerated, locks, arrays, records, and references. The translator converts some of these types of BIR into types of MDG-HDL accordingly to the following table:

BIR Type	MDG-HDL Type
boolean	$conc_sort(bool, [0,1])$
range $x..y$	$conc_sort(\langle$sort name$\rangle, [x, \ldots, y])$
enum $\{A, B, C\}$	$conc_sort(\langle$sort name$\rangle, [A, B, C])$
Array and Record	flattened and declared as signals

where x and y are integers, A, B and C are integers or strings. Note that for range type, we need to enumerate all the values that hold between x and y and for array type, all the signals have the same MDG sort.

A *lock* is an aggregate with the following components:

- An owner variable indicating which thread holds the lock
- A count variable, recording the number of acquisitions of the lock (for relocking)
- A wait variable for each thread indicating whether that thread is blocked in the wait queue of this lock

A *reference* is an aggregate with the following components:

- a refIndex variable, specifying the target pointed to (or 0 for null)
- an instance variable, specifying the instance number of the collection.

Note that reference variables are unusual in that they may appear as final values in expressions even though they are implemented as two variables. The problem can be solved by combining (R_refIndex and R_instNum) into a single integer for the purposes of assignment and equality tests. Dereferencing (the most common operation) uses the components separately.

D. Threads

A thread is specified as set of guarded transformations on the objects. It can be represented as directed graph in which each arc is labeled with a guarded transformation of the form:

$$\langle location \rangle \; : \; \textbf{live}\{\text{set of objects}\} \\ \textbf{when} \; \langle guard \rangle \; \textbf{do} \; \langle action \rangle \; \textbf{goto} \; \langle location \rangle$$

A thread must have at least one location; the first location listed is the start location. Each location is labeled with unique identifier, which is used to refer to the location in a goto. A location may also optionally specify a set of local objects that are live at the location. The guard expression must be boolean expression while the action can be either an assignment, a lock operation, a thread operation, or an assert operation.

The threads are running in parallel but in an interleaved model, i.e, asynchronous. Therefore, the translation schema starts by introducing a concrete variable *runningThread*, which will hold the name of the thread taking step. Thus, it is an enumeration of thread names:

$$conc_sort(ThreadType, [NoThread, T_1, \ldots, T_n]).$$
$$signal(runningThread, ThreadType).$$

Next, we associate with each thread the following variables:

- *location* variable is of type concrete sort and represents the set of locations of the current thread. As indicated above, these locations are uniquely identified.
- *active* variable is a boolean flag that represents the status of the thread and can be changed by the current thread or another one. Typically, the thread actions are (i) *start* action changes the state of another thread from inactive to active, (ii) *exit* action changes the state of the current thread to inactive and (iii) *join* action changes the current thread until another thread becomes inactive (i.e., exits). Note that the active variable is not associated with the *main* thread.
- *blocked* variable is a boolean flag that indicates the current thread is blocked. This may happens when a thread reaches a certain location where the guards are false. The disjunctive boolean expression associated to this flag is derived from the set of guarded transformation of the current thread. In the example above within the Deadlock thread, the blocked variable is defined as:

$$\text{Deadlock_blocked} \triangleq (\text{Deadlock_location} = s_{25} \wedge \text{E_Bool} = 0) \vee \ldots$$

- *terminated* is similar to blocked boolean variable but indicates that the current thread exits. In the example above within the Deadlock thread, the terminated variable is defined as:

$$\text{Deadlock_terminated} \triangleq (\text{Deadlock_location} = s_{44} \vee \ldots)$$

E. Guarded transformation

The thread's set of guarded transformation is processed into three steps:

1) Transitions between locations are mapped into a table which has the following form:

   table([T_location,runningThread,T_active, n_T_location], ...)

   where T_location is the current location and n_T_location represents the next location (i.e, goto location). Note that the *main* thread's table does not contain the active column.
2) Each guard is a boolean expression and is represented as a collection of predefined components.
3) Each action is translated into a table which contains the current location, the runningThread variable, the boolean output of the guard constructed in step 2, the active variable, the state variable and its next state. The table's body is defined according to the action kind described below.

An action is either an assignment, a lock operation, a thread operation, a print operation, or an assert operation. We support only a subset of these actions. Note that print actions are used to output specific values of variables of three types, namely

range, boolean, and enumerated during a simulation. This type of action is ignored.

Another alternative for translating a guarded transformation in a structural style consists in mapping locations to registers and guard-action pairs to multiplexers [6].

F. Properties

A BIR Specification Language (BSL) allows to specify the properties of the program as assertions in certain program locations and/or by using the temporal patterns. As a first step, we are interested only in invariant checking. Within the frames of MDG-HDL, an invariant is mapped to a combinational circuit. The deadlock invariant can be generated automatically from our translation schema by using the boolean variables *terminated* and *blocked* associated to each thread:

$$(\neg T_blocked \wedge \neg T_terminated) \vee (T_terminated)$$

which means that either the thread is taking step or it is terminated. Note this invariant can be generalized to n threads.

IV. Example

a) Simplified Model: As an example, we have chosen a Java implementation of a growable stack[1] The stack is described by (i) a dynamic array of integers, (ii) a constant *size* that specifies the depth of the stack, and (iii) a integer variable *tos* that points to the top of the stack. Moreover, the stack class provides three methods: (i) the *constructor* method that allocates space for dynamic array according to a specified size and initializes the top of the stack (*tos*) to -1, (ii) the *push* method adds a specified item to the stack and increments the top of the stack, and (iii) the *pop* method returns the current item specified the top of the stack and decreases the latter. Moreover, we made *erroneous* modification, namely, we removed from source code a part that allocates extra space for stack when it is full. We do so because we are going to check a property that states that a top of the stack never goes beyond its limits, i.e., the top of the stack holds between its initial value (-1) and the specified size of the stack. Note that this property is added to the source code.

The code of growable stack implementation is used as input file for Bandera tool. Then, we derived a flowchart from BIR specification to make the translation easier. Analysis of the BIR specification flowchart gives us the following results: (i) there are three state variables within this model which are interesting for purposes of the invariant checking, namely $mystack.tos$, i, and i_0. The state variable $mystack.size$ is never changed. So, it can be considered as a constant signal, and (ii) the guarded transformations are summarized in the following table:

Guard	Action
$i < 9$	$tos := tos + 1; i := i + 1$
$i \geq 9 \wedge i_0 < 15 \wedge tos \geq -10$	$tos := tos - 1; i_0 := i_0 + 1$
$i \geq 9 \wedge i_0 < 15 \wedge tos < -10$	$i_0 := i_0 + 1$

[1] javaboutique.internet.com/javasource

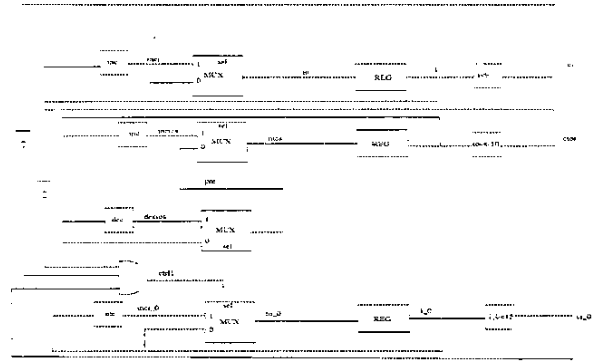

Fig. 3. Sequential circuit derived from BIR Model.

b) Invariant Checking: Bandera extracts the invariant from source code and placed into a file. As result, the invariant to be checked looks like follows:

```
[]((tos > -2) && (tos < size))
```

Then, we develop a combinational circuit that represents the property to be checked to create an invariant specification file. Finally we perform the invariant checking and as expected the invariant does not hold.

V. Conclusion and Future Work

In this paper, we investigated the use of MDG tool in the verification of software models. We have outlined the mapping rules for translating a program model from Bandera Intermediate Representation (BIR) to the MDG-HDL language where a significant subset of BIR is handled. To illustrate the effectiveness of this approach, a simple Java program was processed with Bandera. A model of this program was represented as a BIR specification. Then it was translated into MDG-HDL and was verified then. As a future work, we are planning to improve the translation schema and providing a better support for checking assertions and temporal patterns.

References

[1] J. Baymgartner, T. Heyman, V. Singhal, and A. Aziz. Model Checking the IBM Gigahertz Processor: An abstraction algorithm for high-performance netlists. In Proc. 11th International Conference On Computer Aided Verification (CAV), LNCS, 1633, pp 72–83, Springer-Verlag, 1999.

[2] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In Proc. of the 15 th International Conference on Automated Software Engineering, Grenoble, France, September 2000.

[3] J. Corbett, M. B. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera : Extracting Finite-state Models from Java Source Code. In Proc. of the International Conference on Software Engineering (ICSE), IEEE Press, June 2000.

[4] K. Havelund and T. Pressburger. Model checking Java Programs using Java PathFinder. International Journal on Software Tools for Technology Transfer (STTT), 2(2), 2000.

[5] F. Corella, Z. Zhou, X. Song, M. Langevin, and E. Cerny. Multiway decision graphs for automated hardware verification. Formal Methods in System Design, 10(1), 1997.

[6] A. Gawanmeh, S. Tahar and K. Winter Interfacing ASMs with the MDG Tool. In: E. Boerger, A. Gargantini, E. Riccobene (Eds.) Abstract State Machines - Advances in Theory and Applications, LNCS 2589, pp. 278-292.Springer Verlag, 2003