Translating Java for Multiple Model Checkers: the Bandera Back-End

Radu Iosif, Matthew B. Dwyer and John Hatcliff
Department of Computing and Information Sciences
Kansas State University *

September 2, 2003

Abstract. One approach to model checking program source code is to view a model checker as a target machine. In this setting, program source code is translated to a model checker's input language using a process that shares much in common with program compilation. For example, well-defined intermediate program representations are used to stage the translation through a series of analyses and *optimizing* transformations and target-specific details are isolated in code generation modules.

In this paper, we present the Bandera Intermediate Representation (BIR) – a guarded-assignment transformation system language that has been designed to support the translation of Java programs to a variety of model checkers. BIR includes constructs, such as inheritance, dynamic creation of data, and locking primitives, that are designed to model the semantics of Java primitives. BIR also includes several non-deterministic choice constructs that support abstraction in modeling and specification of properties of dynamic heap structures.

We have developed a BIR-based tool infrastructure that has been applied to develop customized analysis frameworks for several different input languages using different model checking tools. We present BIR's type system and operational semantics in sufficient detail to support similar applications by other researchers. This semantics details several state space reductions and state space search variations. We describe the translation of Java to BIR and how BIR is translated to the input of several model checkers.

1. Introduction

Several research efforts [4, 9, 13, 29, 33, 54, 55] are demonstrating that exhaustive state-exploration techniques such as model-checking can be effective for identifying defects in software that are difficult to find using conventional testing methods.

Tool development efforts in software model-checking have been based on two different architectures. Some have taken an interpretation approach by building a dedicated model checker for a specific programming language. For example, SLAM [4] and BLAST [29] are analysis tools that work directly on C, while Java Path Finder (JPF) [54] works directly on Java bytecodes. Others have taken a translation approach

* `http://www.cis.ksu.edu/bandera`, `{dwyer,hatcliff,iosif}@cis.ksu.edu`, 234 Nichols Hall, Manhattan KS, 66506, USA.

by compiling programs directly into a relatively expressive verifier input language. For example, FeaVer translates C programs into PROMELA, the input language of the SPIN model checker [32], an earlier version of Java Path Finder [28] translated Java to PROMELA, and JCAT translates Java into the input language dSPIN [14] – an extension of SPIN that provides support for programming language features such as dynamic object creation, garbage collection, and method calls.

We have taken the *translation* approach in developing the Bandera tool set because at present it is unclear what collection of state-space representation, reduction, abstraction and search methods are best-suited for model-checking software. In fact, research has shown that changing the computation style or architecture of a particular concurrent program can dramatically impact the relative performance of different state-space exploration techniques; different techniques perform better on different systems [3]. Moreover, if one is interested in experimenting with a new technique on a real programming language like Java, numerous infrastructure components such as parsers, intermediate representations, static analyses, and visualization facilities, are required before one can build a system upon which an empirical evaluation can be carried out. Bandera's architecture cleanly factors out the state-space exploration engine, and allows researchers to reuse all the infrastructure components above by simply plugging in a translation from our Bandera Intermediate Representation (BIR) to the input language of the new state-space exploration engine.

1.1. The Architecture of Bandera

The goal of Bandera is to provide an open infrastructure that allows for easy incorporation and experimentation with multiple analysis and verification techniques. Bandera translates Java source code to a model expressed in the input language of one of several verification tools including SPIN [32], dSPIN [14], HSF-SPIN [19], NuSMV [7], and JPF [54]. The architecture of Bandera shares much in common with that of modern optimizing compilers [43], but it differs in several important respects. Similarities include the staged application of a series of program analyses and transformations, the use of well-defined intermediate program representations to which those transformations are applied, and the isolation of target specific details in code generation modules. The main differences are related to the fact that in a compiler the primary objective is to reduce the run-time of a program, whereas in Bandera, the primary goal is to reduce the amount of memory required to represent the state space of the program since state explosion is the chief barrier to scalability of model checking.

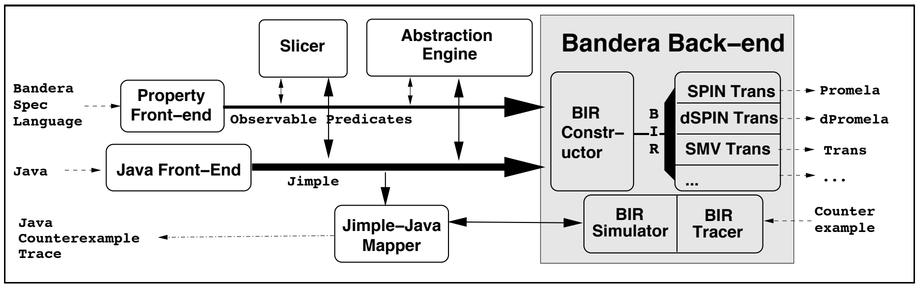

Figure 1. Internal architecture of the Bandera Tool Set

Figure 1 presents the internal architecture of Bandera, and below we briefly summarize the functionality of the components. Bandera is built on top of the *Soot* Java compiler framework [53] developed by Laurie Hendren's Sable group at McGill University. Soot includes an intermediate language called *Jimple* that is a language of control-flow graphs where statements appear in three-address-code form and various Java constructs, such as synchronized statements, are represented in terms of their virtual machine counterparts (such as `monitorenter`, `monitorexit` bytecodes). A Java front-end produces a Jimple representation of the input program.

Source code properties to be checked are written in the Bandera Specification Language (BSL) [11]. BSL consists of a collection of parameterized macros [18] that can be instantiated to different temporal logics, such as linear temporal logic (LTL) [42]. BSL specifications are parameterized by *observables* (predicates on program state) that are defined in Java source code using Javadoc comment notation. A property front-end extracts all the observables declared in the given source program, type checks the declared observables, instantiates the BSL specification to a particular temporal logic, and generates Jimple code that encodes the observables used in the input specification.

Bandera's approach to model construction is to generate one model for each property to be checked. This approach is based on the insight that, given a specific property ϕ, many parts of the software may not influence ϕ at all. Bandera applies model reductions based on the semantics of ϕ to the Jimple representation of the program. Bandera uses both *program slicing* and *data abstraction* (abstract interpretation) to customize models. The Bandera slicer takes as input all the observables mentioned in the input property ϕ and, using an enriched set of program dependences [25], eliminates all Jimple statements that can be shown to not influence the semantics of ϕ's observables [27]. Whereas slicing eliminates both data and control states of a program, Bandera's abstraction component automates support for reducing the number of data states by reducing the size of the data domains over which program variables range [17, 26]. Users select or define predicates over program variables, for example, the data expressions in ϕ's observables, and Bandera automatically synthesizes safely abstracting operator definitions and substitutes those operators into the Jimple program representation.

The Bandera back-end is like a code generator, taking the sliced and abstracted program and producing verifier-specific models. The back-end also functions like a debugger by providing a verifier-independent representation of counter-example information. The back-end components communicate through BIR which acts as an intermediary between the Java-based Jimple representation and verifier-based transition system representations (e.g., Promela). As shown in Figure 1, the back-end has one fixed component called BIRC (Bandera Intermediate Representation Constructor) that accepts Jimple and produces BIR. For each supported verifier, there is also a translator component that accepts the program represented in BIR and generates input for that verifier and a component that translates verifier counter-examples into a trace in the BIR transition system. Translators for SPIN, dSPIN, HSF-SPIN [40], and NuSMV [7] have been built.

1.2. Contributions of this paper

This paper makes two main contributions: (1) we describe several novel BIR constructs that are useful for modeling a variety of software descriptions and argue that support for those constructs would be useful additions to model checker input languages; and (2) we describe the Bandera back-end which is a rich tool infrastructure that applied model checking researchers can exploit to quickly develop model checking frameworks for software design and implementation notations.

Since we intend for this paper to serve as a reference for those interested in writing translations to and from BIR, we devote significant attention a formal presentation of the semantics of BIR. We believe that this formal presentation clarifies a number of subtle semantics issues associated with BIR that might otherwise be ambiguous.

BIR has evolved based on lessons learned from developing translations from programming languages (e.g., Ada and Java) to model checker inputs (e.g., Promela, SMV TRANS systems). BIR is designed as a guarded-command language to be similar to the input languages of existing model-checkers so that translations to existing model-checking tools can be written with minimal effort. It includes several features that we believe are essential for effectively supporting model checking of software systems.

1.2.1. *Novel Primitives for Modelling Software*

BIR provides built-in support for Java language features to facilitate translations from Java/Jimple into BIR. Rather than present these features at the level of granularity found in source languages, such as Java, we have developed finer grain support that is amenable to translation to a broader set of target model checkers and allows translations to minimize the state-space based on a program's usage of language features. Specifically, BIR supports fine grained locking primitives that can be composed to implement Java synchronized statements and wait-/notify synchronization. BIR provides flexible mechanisms for defining state variables including type and allocator-specific data collections for storing anonymous data to model heap-allocated data in programs. Translators from BIR to model checker input languages can exploit the semantics of these primitives to optimize the transitions and states that encode this information.

1.2.2. *Rich Forms of Non-deterministic Choice*

BIR incorporates operators for modeling forms of non-deterministic choice that are essential for defining a rich class of program properties and abstracted programs. Specifically, BIR supports non-deterministic choice over heap-allocated data modeled by BIR collections. Primitives are defined for non-deterministically selecting an allocated instance of a given type in a program state and for non-deterministically selecting an instance that is reachable from a given reference in a program state. We have found these to be essential for expressing program properties related to the structure of heap-allocated data [11], for modeling abstractions of program data structures [17], and for defining *environments* that model the behavior of hardware and software components that are external to the system under analysis [51, 52]. A further refinement of the non-deterministic choice primitives in BIR is to distinguish those that are used for modeling environments from those used to model the system under analysis. The distinction between *internal* and *external* choice can be exploited to perform efficient searches of abstracted state spaces [48].

To support the effective use of BIR and the Bandera back-end facilities we describe the features and semantics of the BIR language and discuss the strategies that developers should follow when (a) translating Java and other design notations into BIR, and (b) translating BIR to input languages of model-checkers and other verification/analysis tools. Specifically, we present the BIR intermediate language, describe how Bandera translates Java into BIR, describe how Bandera translates BIR into Promela and outline general strategies that developers should follow when translating BIR to other model-checker input languages.

Finally, we give an overview of the semantics of BIR and address subtle issues regarding the translation of BIR's virtual coarsening and non-deterministic-choice constructs. To supplement this presentation, the Bandera Project web site `http://www.cis.ksu.edu/bandera` provides the Bandera open-source distribution, user's manual, and an example repository. In particular, a *BIR Back-end Developers Kit* is available which provides the BIR parser, source code for BIR back-end translators to illustrate translation techniques, and documentation.

1.3. Organization

The rest of this paper is organized as follows. Section 2 introduces a Java example that we will use to illustrate the principles for translating Java to BIR and then translating BIR to model-checker input languages. Section 3 outlines the Java to BIR translation, while Section 4 outlines the translations from BIR to model-checker input languages like Promela – the input language of the SPIN model-checker. Section 5 gives a formal presentation of the novel features of BIR. Section 6 presents related work, and Section 7 concludes.

2. Example

This section introduces an example that will be followed throughout the paper in order to show how Java programs are translated by Bandera into finite-state models. The program fragment in Figure 2 illustrates the implementation of a message dispatcher that enables communication between an arbitrary number of clients and servers. Messages are instances of a class `Msg`, containing priority numbers as illustrated in Figure 3. Messages are produced by client threads and sent to the message queue using its `send` method. The implementation of the dispatcher ensures that the messages will be received in priority order. The `MsgQueue` class is designed to be thread-safe, as the `send` and `recv` methods use the common `wait-notify` synchronization coding pattern.

To formalize the prioritized receipt of messages, we declare a set of atomic propositions, encoded as BSL observables `called(this, msg)` and `returns(this, msg)`. The BSL predicate `called(this, msg)` holds when control is at the first line of the method `send` and the reference value bound to the predicate parameter `msg` equals the reference value of the method parameter `m`. The BSL predicate `returns(this, msg)` holds when control is immediately after any `return` statement of method `recv` and the reference value bound to the predicate parameter

```
    class MsgQueue {
[1]   Msg tail;
[2]   int max, no;
[3]   MsgQueue(int max) {
[4]     this.tail = null;
[5]     this.cap = max;
[6]     this.no = 0; }
/**
 * @observable
 * INVOKE called(this, Msg msg):
 *              msg == m;
 */
[7]  synchronized void send(Msg m) {
[8]    while (no == max) {
[9]    try { wait(); }
[10]   catch(...) { return; }
[11] }
[12] Msg curr = tail;
[13] Msg last = tail;
[14] while (curr != null &&
[15]    curr.prio >= m.prio) {
[16]    last = curr;
[17]    curr = curr.next; }
[18] if (last == null ||
[19]     curr == last)
[20]   tail = m;
[21] else
[22]   last.next = m;
[23] m.next = curr;
[24] no ++;
[25] notifyAll(); }
/**
 * @observable
 * RETURN returns(this, Msg msg):
 *              $ret == msg;
 */
[26] synchronized Msg recv() {
[27]   while (no == 0) {
[28]     try { wait(); }
[29]     catch(...) { return null; }
[30]   }
[31]   Msg m = tail;
[32]   tail = tail.next;
[33]   no --; \\[33]
[34]   notifyAll();
[35]   return m; }
```

Figure 2. Message Queue Example

msg equals the return value of the method. In addition, we consider the predicate higher(m_1, m_2) which formally encodes the m1.prio > m2.prio condition. Having these predicates, we can use BSL to state that for all instances of the Msg and MsgQueue classes, whenever messages m_1 and m_2 are to be returned from a given queue, it must be the case that the higher priority message is returned before the lower priority message. Without going into all the details (the reader is referred to [11]), we simply note that the resulting BSL specification can be mapped down to the following enhanced LTL formula:

$$\forall q : Q, m_1, m_2 : M, m_1 \neq m_2 \; [\Box((\Diamond returns(q, m_1) \wedge \Diamond returns(q, m_2)) \\ \Rightarrow (higher(m_1, m_2) \iff !returns(q, m_2) \; U \; returns(q, m_1)))]$$

Here, Q and M denote the finite sets of instances of the MsgQueue and Msg classes, respectively that are used to represent BSL universal quantification. Universal quantification in a BSL specification binds each allocated instance of a designated type (e.g., MsgQueue) to a named variable (e.g., q), and then checks the temporal specification with those bindings. The predicates *returns* and *higher* are evaluated on the instances bound by the quantifications. This requirement captures an important aspect of correct message queue behavior.

Another correctness issue is related to the *fairness* of the dispatcher. In our implementation (see Figure 2) messages with lower priorities can

```
/**
 * @observable
 * static EXP higher(Msg m1,
 *                   Msg m2):
 *          m1.prio > m2.prio;
 */
class Msg {
  Msg next; int prio;
  Msg(int p) { this.prio = p; }
  }
class DataMsg extends Msg {
  int data;
  DataMsg(int p, int d) {
    super(p); data = d;
  }
  int get() { return data; } }
class RequestMsg extends Msg {
  char req;
  RequestMsg(int p, char r) {
    super(p); req = r;
  }
  char get() { return req; } }
```

Figure 3. Data and Requests

be forever neglected. This issue can be addressed by using the appropriate LTL fairness requirement as an assumption for other correctness properties[1].

In order to verify the `MsgQueue` class with respect to its specifications we complete the implementation with the code of client and server threads and a main method that instantiates several such threads as shown in Figure 4. Clients internal decisions about sending data (`DataMsg`) and request (`RequestMsg`) messages to servers is abstracted using non-deterministic choice (`choose()`). Both `DataMsg` and `RequestMsg` are subclasses of `Msg` as shown in Figure 3. Upon receiving a message, the server will extract the message and process it depending on the dynamic type of the message; for this example, requests indicate whether subsequent integer data should be added to or subtracted from a running total.

The sample Java code from Figures 2, 3, and 4 uses the following language features: dynamic creation of objects and threads, monitor-based and condition-based synchronization, and inheritance and dynamic type lookup. In the remainder of the paper, we will focus on these aspects of Java programs, while describing the design of BIR, and translations to and from BIR.

3. Translating to BIR

The core of BIR is a guarded assignment language and as such it can model a wide variety of state-based system descriptions. We have developed translators for several such description languages in addition to Java, including statecharts [37] and synchronization policy speci-

[1] To verify a property P under the fair dispatcher assumption we verify the property: $\forall q : Q, m : M\ [\Box((\mathit{called}(q,m) \Rightarrow \Diamond \mathit{returns}(q,m)) \Rightarrow P)]$

```
public static void main(...) {
 MsgQueue q = new MsgQueue(10);
 Server s = new Server(q);
 (new Client(q,1)).start();
 (new Client(q,2)).start();
 s.start(); }
class Client extends Thread {
 MsgQueue q; int p;
 Client(MsgQueue q, int p) {
  this.q = q; this.p = p;
 }
 public void run() {
  int i;
  while (true) {
   Msg m;
   if (choose())
    m = new RequestMsg(p,
        choose() ? "+" : "-"));
   else
    m = new DataMsg(p,i++));
   q.send(m);
  }
 } }
```

```
class Server extends Thread {
 MsgQueue q;
 Server(MsgQueue q) {
  this.q = q;
 }
 public void run() {
  int total:
  boolean lastAdd = true;
  while (true) {
   Msg m = queue.recv();
   if (m instanceof DataMsg)
    if (lastAdd)
     total += ((DataMsg)m).get();
    else
     total -= ((DataMsg)m).get();
   else
    lastAdd =
     ((RequestMsg)m).get()=='+');
 }
} }
```

Figure 4. Sample Client and Server

fications [15]. To effectively model Java programs, BIR has been designed to include primitives for modeling object-oriented, dynamism and concurrency features that are specific to the JVM [39]. In this section, we describe how these features are translated from a Java/Jimple representation of a program to BIR.

Prior to the Jimple to BIR translation implemented in the BIRC component, our current tools perform two Jimple transformations: virtual call resolution and method inlining. Virtual call resolution determines the possible receiver types at a method call site via class hierarchy analysis [12] and introduces explicit type tests to guard calls to the appropriate method for the tested type. This enables inlining of methods since the guards ensure that a single receiver type reaches each call site. Inlining is then performed with appropriate renaming of local variables and mapping of actual parameters and return values to formals. Ongoing work on Bandera is adapting BIR for model-checkers such as dSPIN [35] that can model virtual method invocation; this will allow treatment of recursive methods.

Much of our Jimple to BIR translation is analogous to well-understood code-generation techniques from program compilation. Unlike traditional compilers, however, we exploit the fact that the entire program is available during translation. This allows us to optimize the generated BIR transition system so that it only models program components that are potentially used during some program run. For example, data that a JVM associates with each Java object in order to implement locking and the semantics of `wait-notify` is only generated for types whose instances are actually locked or on whom `wait` or `notify` is called. Similarly, storage for instances of classes is allocated only for those classes that appear in `new` statements. Thus in our example no storage will be allocated to store instances of class `Msg` since no `new Msg()` expressions appear in the program. This helps to minimize the size of each program state that is explored during model checking.

Our translation treats basic Java library classes, such as `java.lang.Object`, `java.lang.Thread` and interface `java.lang.Runnable`. Reference to an instantiation of these classes is supported in a limited form. Specifically, methods `start()`, `exit()`, `run()`, and field `target` of `java.lang.Thread`, and `wait()`, `notify()`, and `notifyAll()` of `java.lang.Object` are mapped to appropriate BIR representations. Other library code can be used, but it must be explicitly included in source code form as part of the program and any native method calls must be replaced with pure Java code.

Our current translation approach has several limitations. Floating point types are maximally abstracted by transforming all test expressions over floating point values to non-deterministic choice over a boolean domain. Recursion and user thrown exceptions are not supported in the current version of our tools. These issues are subject to ongoing work. Some methods of basic library routines, such as `getClass()`, `hashCode()`, `clone()`, `finalize()`, and timed versions of `wait()` in `java.lang.Object` are not supported, neither is program input or output.

Our presentation is driven by identifying extracts of the example from Section 2 and describing the corresponding fragments of BIR. The BIR fragments have been modified to improve their readability by shortening variable names and eliding details. Temporary variables that model JVM stack locations are named with `tmp_` prefixes. We begin with an overview of BIR which at the highest level of structure has two parts: (1) a passive part that declares the data layout of the system, and (2) an active part that declares the threads of control and transformations of the system. The syntax of BIR is given in Appendix A.

Passive BIR Declarations: Typically, the data declaration section will describe a bounded data space by bounding both basic data types (e.g., integer values are bounded by subranges) and dynamically allocated data (e.g., objects are allocated from pools of bounded size). However, when generating BIR system descriptions for translation to model-checkers that do not require such bounds (e.g., dSPIN supports dynamic object creation and garbage collection directly), they can be omitted as appropriate.

BIR provides four categories of types. *Primitive* types include boolean, integer subranges, and enumerated types. *Lock* types are used to implement thread synchronization. *Aggregate* types include records and arrays. *Reference* types are pointers to aggregate types. BIR's type-checking strategy for records and enumerated types is similar to C/C++ in that it is based on name-equivalence instead of structural-equivalence [21].

A reference type declaration includes the type of objects to which the reference can refer, and a list of collections that can hold objects to which the reference can refer. Supplying an object type in a reference type declaration allows type-checking to easily produce a static type of an object returned by a dereference expression (*à la* Java). Supplying a collection list allows back-end translators to produce more efficient procedures for object dereferencing and enables optimizations based on (non)aliasing information.

Variables of `lock` type are used to represent the implicit lock field associated with each Java object. In Java, locks can be *reentrant* (i.e., acquired more than once by the same thread) [39], and threads can also *wait* (i.e., suspend themselves) on a lock. Extra state data is required to maintain information about reentrant locks and locks upon which `wait()` is invoked. If static analysis determines that an object's lock is not reentrant or not involved in a `wait()`, those qualifiers can be removed from the lock variable's type. This mechanism informs back-end translators that unnecessary state components in a lock's representation can be omitted.

To carry out appropriate type-checking and to implement Java operations such as `instanceof`, type casts, and virtual method invocations, BIR allows declaration of an inheritance hierarchy which gives rise to a subtyping relation; to accommodate a variety of source languages, BIR supports any acyclic subtyping relation. Any type identifier that appears in the inheritance hierarchy declaration must be bound to a record type specification.

Collections provide a flexible representation of the heap in several ways. First, they allow alternative heap representations depending on the target model-checker. For example, when translating to a model-checker like SPIN which does not provide any built-in symmetry reductions, using a different collection for each allocator site in a Java program achieves a simple but effective form of symmetry reduction (explained later in this section). However, when translating to dSPIN which provides built-in heap symmetry reductions, it is more effective to use a single collection for each Java class. Second, the collection representation allows heap data to be bounded in a flexible way (explained in Section 4).

Active BIR Declarations: Thread declarations are used to define independently executing transition systems. In each thread, declarations of local variables are followed by a sequence of *locations*. Each location has an associated *live clause*. The meaning of the live clause is to specify which variables are live at that particular control point. This information is useful in reducing the size of the generated transition system, by resetting dead variables to zero. When system execution begins, control in each thread begins at the *start location* – the first location in the thread's location sequence. At any given time, a thread is at one of its locations, called the *current location* of the thread.

Each location is the source of one or more *guarded transformations*. Each transformation consists of a boolean *guard expression* followed by a sequence of *actions* ending in the target location indicating the source of the next transformation in the thread to be executed. To generate a successor of a given state, a transformation whose guard is true and whose source location is the current location of its thread is selected. The transformation's actions are executed sequentially, updating the system state (atomically) to produce the next state. Transformations may be annotated as `invisible` indicating that it is safe to collapse the transformation along with its successor into a single atomic step.

Lock and thread operations must appear in certain patterns (the Java compiler and BIR constructor can easily guarantee this, but other translators generating BIR should observe these constraints). Specifically, *lock* and *unlock* operations must be properly nested and a thread must never attempt to *lock* a lock that it already holds unless the lock is declared reentrant. Each *lock* operation must be guarded by a *lockAvailable* test, and each transformation corresponding to code that occurs inside a synchronized block must be guarded by a *hasLock* test. A *wait* operation must be the last action of a transformation and must be followed by an *unwait* operation that is guarded by both a *lockAvailable* test and a *wasNotified* test. The purpose of these last two restrictions is to leave hooks for translators so they can implement the monitor semantics in the most efficient way. For example, in SPIN it is better to prevent a thread from executing *lock* until the lock is available, while in NuSMV it is better to allow the thread to execute *lock* (unsuccessfully), but then wait until the lock is released and given to the thread (by the releaser).

Most other expressions appearing in actions and guards are conventional. Some exceptions include the `externChoose` and `internChoose`

```
loc s123: live { m, last, curr, ... }
  when (last == null) do invisible { } goto s124;
  when (! (last == null)) do invisible { } goto s143;
loc s124: live { ... }
  when (curr == last) do invisible { } goto s125;
  when (! (curr == last)) do invisible { } goto s143;
loc s125: live { ... }
  when true do { this.tail := m; } goto s126;
```

Figure 5. Basic Transformations

constructs, each of which represents non-deterministic choice over the values in the argument list. The `externChoose` is used to represent non-deterministic choice in the environment component (e.g., a test harness for the system), whereas `internChoose` is used for non-deterministic choice in the system itself (Bandera's abstraction facilities use it to represent data abstractions). These choice constructs are handled differently in the *choose-bounded* search strategy described Section 5.0.3. In addition to choosing over a fixed set of values, BIR also includes expressions for non-deterministically choosing from the allocated instances of a collection and for choosing from the instances reachable from a given reference in the current heap state. These expressions have been used to state program properties related to the heap [11] and in developing abstract models of the environment [50]; their semantics is discussed in detail in Section 5.0.2.

Basic Transformations and Visibility: Expressions and statements that treat JVM base types have a natural mapping to BIR. The BIR fragment in Figure 5 illustrates how the compound test on lines [18-20] of Figure 2 is translated to a series of guarded transformations in BIR and how a field assignment is expressed directly.

We note that transformations involving only locals (in this fragment of `MsgQueue.send()` variables `m`, `last`, `curr` are all locals) are marked as invisible to indicate that their effect can only be observed by the containing thread. In contrast, the assignment at location `s125` is observable since it corresponds to a write to heap allocated data.

Inheritance: Inheritance is present in our example via subtyping of `Msg` by `DataMsg` and `RequestMsg`. We illustrate the modeling of subtyping relations in BIR in Figure 6. The record types `RequestMsg` and `DataMsg` explicitly represent the inheritance of fields `next` and `prio` from `Msg`. The reference types indicate the collections whose elements may be referenced by a value of the type. `Msg_ref` is defined to reflect

```
system MessageQueueExample()
 Msg_ref = ref Msg { RequestMsg_col, DataMsg_col };
 RequestMsg_ref = ref RequestMsg { RequestMsg_col };
 DataMsg_ref = ref DataMsg { DataMsg_col };
 Msg = record { next : Msg_ref; prio : range -1..3; };
 RequestMsg = record { next : Msg_ref; prio, req : range -1..3; };
 DataMsg = record { next : Msg_ref; prio, data : range -1..3; };
 DataMsg extends Msg;
 RequestMsg extends Msg;
 Msg extends Object;
 RequestMsg_col : collection [3] of RequestMsg;
 DataMsg_col : collection [3] of DataMsg; ...
loc s70: live { m, ... }
 when true do invisible
  { tmp_9 := (m instanceof DataMsg); } goto s71; ...
```

Figure 6. Inheritance

the fact that a Java variable declared of `Msg` can refer to an instance with dynamic type `DataMsg` or `RequestMsg`. Finally, the subtyping relationships among records is explicitly defined by the `extends` clause, since subtyping in Java is by name rather than structural. Location `s70`, which models the conditional expression on line `[84]` in Figure 4, illustrates a guarded assignment that uses one of BIR's JVM specific operators; `instanceof` in BIR has the same semantics as in the JVM.

Heap Allocated Data: Data in Java programs is either global, stack or heap allocated. Inlining effectively flattens stack allocated data associated with called methods and models it as local data in the calling BIR thread. BIR's collection facility provides a flexible mechanism for modeling heap allocated data. A collection is, in essence, a typed array of records, and we model the global program heap as a group of collections. Rather than use a single collection for each Java class, we introduce a collection for each allocator of a class (i.e., `new` expressions). Figure 6 illustrates the collections generated for the two allocation sites (on lines `[67]` and `[70]`) in the `Client.run()` method of Figure 4.

In the presence of multi-threading, this heap modeling provides a simple form of heap symmetry reduction by allocating instances in a collection in an order that is determined locally by a thread's behavior. A single collection per type would introduce allocation orders that depend on the interleaving of threads performing the allocations.

Resource Bounds: To enable efficient reasoning, Bandera allows users to define bounds on the range of values that program data can take on. Figure 6 illustrates the modeling of integer fields `prio` and `data`, for the

```
process MessageQueueExample()
 Server = record { tid : tid; queue : MsgQueue_ref; }; ...
main thread Main()
 s : Server_ref := null; ...
loc s10: live { q }
 when true do invisible { s := new Server_col; } goto s11; ...
loc s43: live { s }
 when true do { s.tid := start Server(s); } goto s44; ...
thread Server(this : Server_ref) ...
```

Figure 7. Thread Creation

fields of `DataMsg` instances, as BIR range types; the default range type is the interval $\{-1, \ldots, 3\}$, but this can be set by the user. Bounds on the number of instances created at an allocator site can also be defined as illustrated in the collection sizes in Figure 6; the default allocation bound is 3, but this can be set on a per class basis by the user. Resource bounds are exploited in performing customized state-space searches as described in Section 5.0.3

Note that when translating BIR to model-checkers that support garbage collection such as dSPIN, the bounds on collections may be ignored as explained in Section 4.2.

Thread Primitives: Java threading primitives are supported directly in BIR. Instances of subtypes of `java.lang.Thread` or classes implementing `java.lang.Runnable` are modeled with both a data and a control component. The data component is a record instance that stores the member data for the class instance. Figure 7 illustrates the BIR fragment that models a `Server` with a `queue` component modeling the field declared on line `[75]` in Figure 4 and with a `tid` field that records the BIR thread identifier for the thread's control component. The predefined BIR type `tid` exclusively specifies thread identifier values.

The control component is derived from the `run()` method for the object, in this case from `Server.run()`. This method is modeled using a BIR thread parameterized by a reference to the data component for the object. This allows access to instance data through references to `this` in the thread body. Thread instances are allocated as shown in location `s10` of Figure 7, and their execution starts after the BIR `start()` operation is called with the thread's data component reference, as shown in location `s43` of Figure 7. The start method returns the thread identifier for the new thread which is stored in the thread record's `tid` field to achieve cross-referencing between data and control components.

```
loc s51: live { this_MsgQueue, ... }
 when lockAvailable(this_MsgQueue.BIRLock) do {
  lock(this_MsgQueue.BIRLock); } goto s52;
loc s52: live { this_MsgQueue, ... }
 when hasLock(this_MsgQueue.BIRLock)
  do { tmp_9 := this_MsgQueue.elements; } goto s53;
loc s53: live { this_MsgQueue, tmp_9 , ...}
 when (tmp_9 == 0) do invisible { } goto s54;
 when (! (tmp_9 == 0)) do invisible { } goto s57;
loc s54: live { this_MsgQueue, ... }
 when true do { wait(this_MsgQueue.BIRLock); } goto s55;
loc s55: live { this_MsgQueue, ... }
 when (lockAvailable(this_MsgQueue.BIRLock) &&
      wasNotified(this_MsgQueue.BIRLock))
  do { unwait(this_MsgQueue.BIRLock); } goto s56;
loc s56: live { this_MsgQueue, ... }
 when true do { tmp_9 := this_MsgQueue.elements; } goto s53;
loc s57: live { this_MsgQueue, ... }
 ...
loc s66: live { this_MsgQueue, ... }
 when true do { notifyAll(this_MsgQueue.BIRLock); } goto s67;
loc s67: live { this_MsgQueue, ... }
 when true do invisible { ret := m; } goto s68;
loc s68: live { this_MsgQueue, ret, ... }
 when true do { unlock(this_MsgQueue.BIRLock); } goto s69;
```

Figure 8. Synchronization

Synchronization Primitives: BIR is designed to support synchronization primitives that closely match those available in Java.

Java synchronized statements are represented as a pair of JVM `monitorenter` and `monitorexit` bytecodes. BIR decomposes the functionality of those operations still further via its `lock` primitives, as discussed earlier, and our translation uses these primitives to achieve Java's monitor functionality.

Locations `[51-52]` in Figure 8 implement the entry of synchronized method `recv()` on line `[26]` in Figure 2. This is achieved in three steps: (1) waiting until the desired lock is available, (2) acquiring the lock via a call to `lock()`, and (3) proceeding into the synchronized region if `hasLock()` is true. Exiting a synchronized region is achieved with a single BIR `unlock()` operation as shown at location `s68` of Figure 8.

Locations `s53-s57` illustrate how we use BIR to model the standard conditional `wait` coding pattern that is common in Java (this pattern is used on lines `[27-30]` of Figure 2). The semantics of Java's `wait()`

```
loc s96: live { ... }
 when true do invisible { tmp_prio := externChoose(0,1); } goto s97;
loc s97: live { tmp_prio, ... }
```

Figure 9. Non-deterministic Choice

operation is achieved by a sequence of three BIR operations: first the thread indicates it wants to `wait()` on the BIR lock, then the thread waits until the both the lock is available and the lock has been notified, it then indicates that it is no longer waiting via the `unwait()` operation. BIR's primitives for `notify()` and `notifyAll()` match the semantics of Java methods exactly; the latter is illustrated in location `s66` of Figure 8.

Non-deterministic Choice: Bandera may introduce non-deterministic choice operators into the program to encode abstractions. Users may also introduce choice operators into their programs as a modeling primitive. The `Client` threads that form the environment of the `MsgQueue` use the boolean `choose()` operator on line `[66]` of Figure 4 to model the lack of knowledge of the specific conditions under which a `DataMsg` or `RequestMsg` may be sent. As described earlier, *internal* choice is used for modeling abstractions. The use of the `choose()` operator, described above, is mapped to external choice and expressed using the `externChoose(0,1)` operator in the BIR fragment from Figure 3.

BSL Predicates: The observable predicates that are used to express properties in BSL must also be expressed in terms of the BIR model. For predicates that are parameterized by quantified variables, as in the properties in Section 2, we translate the predicates into a form that explicitly refers to BIR variables that hold bound values. In general, predicates, such as the one preceding line `[7]` in Figure 2, may refer to a control location, such as the invocation of method `send()`, and a data constraint, such as a test for equality between the second predicate parameter and the `Msg` parameter of the `send()` call. The BIR predicate is as follows:

```
  pred_called = (Client(null)@s125 &&
               ((quantification_m1 == Client:send_m) &&
                (quantification_mq == Client:send_this)));
```

where a location `s125` is the call of `send()` from in a `Client` thread. The `null` parameter in location predicate indicates that the `tid` of the thread is unconstrained; it will be true if any instance of `Client` reaches location `s125`. Names of the form `quantification_` refer to

BIR globals bound to quantified values, and names of the form `Client:` indicate method locals or parameters.

3.1. Counter-example Interpretation

Just as BIR insulates source-language concerns from verifier concerns in the generation of model checker inputs, it also insulates clients from needing to build counter-example processing capabilities for model checker specific counter-example formats. The BIR back-end supports this by requiring that BIR-to-verifier translators include a component that maps verifier-specific counter-examples back to a BIR trace. A BIR trace is a finite-sequence of BIR transformations that can be used to generate the state information on any prefix of the trace. For transformations that correspond to non-deterministic choice expressions additional information defining the *chosen* value is encoded in the trace.

Back-end clients interact with counter-examples through a BIR simulator, illustrated in Figure 1. The simulator provides basic capabilities for stepping forward and backward through a trace and for querying the values of state variables at a given point in the trace. These capabilities have been used to build Java-specific debugger-like facilities for exploring counter-examples in Bandera [10] and for animating counter-examples on visual depictions of statecharts [37].

4. Translating BIR to Model Checker Inputs

This section is dedicated to the model generation techniques used in Bandera. We focus mainly on the description of the translator to the SPIN model checker [30]. Translation to the dSPIN [35] model checker is discussed briefly and the translation strategy for NuSMV [6] is presented in Appendix C. Throughout this section, we denote by *model* the description of a program's behavior, rather than the subset of behaviors satisfying a temporal logic specification.

4.1. The SPIN Translator

The following discussion assumes a certain degree of familiarity with Promela [30], the input language of the SPIN model checker. We present informally the translation scheme for the most relevant primitives in BIR, such as dynamic object creation and synchronization actions. The translation to SPIN also supports dynamic thread creation, which relies on the underlying support of SPIN for dynamic processes.

Object Creation: Object allocation is modeled in Promela using collection variables, declared within the state-vector. For instance, the collection of three elements of type `RequestMsg` from Figure 6 is translated into the following structure:

```
typedef type_24 { bit inuse[3]; RequestMsg instance[3]; }
```

Here the `inuse` bit-vector marks collection slots that have already been allocated, while the `instance` vector stores the instance data.

Heap allocated data is accessed in BIR via reference values. In the Promela model we represent a reference value by a two-byte integer, where the most significant byte uniquely identifies the collection, and the least significant byte is the index of the instance within the collection. References are created by the `_ref` macro, and accessed by the `_collect` and `_index` macros. Allocation itself is performed via the `_allocate` macro whose definition is given below:

```
#define _allocate(col, refindex, maxsize, locNum, transNum, actionNum)
do
:: col.inuse[_i_] ->
   _i_ = _i_ + 1;
   if  :: _i_ == maxsize -> printf("BIR: ... LimitException\n");
          limit_exception = true; _i_ = 0; goto endtrap;
       :: else
   fi;
:: else -> col.inuse[_i_] = true; _temp_ = _ref(refindex,_i_); _i_ = 0;
   break;
od
```

The first available collection slot is searched. In case one is found, it is marked `inuse` and a new reference value is created from the collection identifier and the current slot index. This value is assigned to a special temporary variable `_temp_` from which it is subsequently read by the program. On the other hand, if the collection is exhausted, then a `limit_exception` is raised by setting a flag and jumping to the `endtrap` location. Exception handling will be discussed in the following.

In both cases the `_i_` counter is dead at the end of the loop, therefore it is reset. As an example, the allocation action occurring at location `s10` in Figure 7 is translated in Promela as follows:

```
loc_10: atomic { _allocate(Server_col,4,3,26,0,1);
                 s = _temp_; _temp_ = 0; ... }
```

To model accesses and updates of dynamic allocated data, the SPIN translator uses *points-to* information to determine the appropriate collection, based on all possible types of a reference variable. For instance, the assignment occurring at location `s125` in Figure 5 is translated as follows:

```
if
:: (_collect(send_MsgQueue_this) == 1) ->
   MsgQueue_col.instance[_index(send_MsgQueue_this)].tail
      = send_MsgQueue_m;
:: else -> printf("BIR: NullPointerException\n"); assert(false);
fi;
```

In this case there is only one collection of type `MsgQueue`, whose index is `1`. An attempt to access a reference variable that hasn't been previously assigned a valid reference value is captured by the `else` branch of the conditional. The effect in this case is to signal a null pointer exception and stop the model checker.

Synchronization Primitives: A (waiting and reentrant) lock object is modeled in Promela by the following structure:

```
typedef lock_RW {
  chan lock = [1] of { bit };
  byte owner, count;
  int waiting;
};
```

The first field is a blocking communication channel defined to hold one (bit) token. Intuitively, an empty channel represents a taken lock. The `owner` and `count` fields are introduced to support reentrant locking, while the `waiting` field is used for waiting and notification primitives. The reentrant locking/unlocking primitives are implemented by the following macros. A formal definition of these operations is given in Section 5.

```
#define LOCK 0
#define _lock_R(sync)
  if
  :: sync.owner == _pid -> sync.count ++;
  :: else -> sync.lock ? LOCK; sync.owner = _pid;
  fi
#define _unlock_R(sync)
  if
  :: sync.count > 0 -> sync.count --;
  :: else -> sync.owner = 0; sync.lock ! LOCK;
  fi
#define _lockAvailable_R(sync) (nempty(sync.lock) || sync.owner == _pid)
```

The first time a `lock` action is performed by a thread on a lock object, the `LOCK` token is removed from the channel. Subsequent `lock` actions by the same thread are non-blocking, the only effect being to increment the lock counter, while other threads will block attempting to receive the `LOCK` token from the channel. Dually, an `unlock` action will release the lock, by sending the token to the channel, only at the outermost level, when the value of the counter is zero. The `lockAvailable` predicate returns true if either the lock channel is not empty or the lock is held by the same thread. We remind the reader that the Promela keyword `_pid` evaluates to the index of the current thread.

The waiting primitives are implemented by the following macros:

```
#define _wait_R(sync)
  if
  :: sync.owner == _pid ->
       sync.waiting = sync.waiting | (1 << _pid); _temp_ = sync.count;
       sync.count = 0; sync.owner = 0; sync.lock ! LOCK;
  :: else -> printf("BIR: IllegalMonitorStateException\n"); assert(false);
  fi
#define _unwait_R(sync) sync.lock ? LOCK; sync.owner = _pid;
                        sync.count = _temp_; _temp_ = 0
#define _wasNotified(sync) !(sync.waiting & (1 << _pid))
```

The `waiting` field of the synchronization structure represents the set of threads that have already performed a `wait` action on behalf of the lock object and are still dormant. The number of (reentrant) lock actions already performed by the waiting thread is recorded into the (local) `_temp_` variable. Finally, the `owner` field is reset to zero and the token is sent to the `lock` channel in order to free the lock object. Attempting to perform a `wait` action on a lock not owned by the thread raises an `IllegalMonitorStateException`.

The `unwait` action is the converse of `wait`, therefore it performs all operations needed by the thread to re-aquire the lock. The `wasNotified` predicate is implemented as a membership test on the `waiting` bitset.

Notification is implemented by the `notify` macro. If there is at least one thread in the `waiting` set, one thread is randomly chosen and eliminated from the set. The non-determinism is captured by the innermost `if-fi` construct. Moreover, if there is at least one waiting thread, it is guaranteed that one will be chosen for notification. If there are no waiting threads, the notification action has no effect. As with `wait`, attempting to notify a lock not owned by the current thread raises an `IllegalMonitorStateException`.

```
#define _notify(sync)
do
:: (sync.owner == _pid) && (sync.waiting != 0) ->
   do
   :: (_i_ < MAXTHREADS) ->
      if
      :: (sync.waiting & (1 << _i_)) -> _temp_ = _i_;
         if
         :: sync.waiting &= ~(1 << _i_); _i_ = 0; _temp_ = 0; break;
         :: else -> skip;
         fi
      :: else -> skip;
```

```
        fi;
        _i_ = _i_ + 1;
      :: else -> sync.waiting &= ~(1 << _temp_); _i_ = 0; _temp_ = 0; break;
      od;
      break;
  :: (sync.owner == _pid) && (sync.waiting == 0) -> break;
  :: else -> printf("BIR: IllegalMonitorStateException\n"); assert(false);
  od
```

As most of the synchronization models involve more than one transition, these macros need to be used only inside `atomic` sequences, in order to guarantee the correct semantics of their executions.

Atomic Sequences: The granularity of a generated model is an important factor that controls the complexity of the verification process. Coarser models are easier to verify, however care must be taken to preserve the semantics of the original program. In Java bytecode, the basic measure of granularity is the JVM instruction. We can generalize this to Java source code, considering in addition that all accesses to the local stack of a thread are invisible to other threads. Since local actions are globally independent [31], executing them without interleaving with other threads is a conservative approach to reducing the size of the state space.

Visibility information is already available in a BIR specification, as every invisible transformation can be annotated accordingly. A sequence of successive invisible transformations, with no intermediate branching, ending with a visible transformation, is translated into a Promela `atomic` sequence. An `atomic` sequence is executed by the model checker without interleaving with other processes.

Another optimization is achieved by resetting the values of *dead* variables. A variable is said to be dead at a program point if, on all control paths starting from that point, any read of the variable is preceded by an assignment to it. When a variable becomes dead it can safely be reset, avoiding the exploration of states that differ only by values of dead variables. In our translation to SPIN, all dead variables are reset at the end of an atomic sequence.

Bounded State Exploration: There are several actions that cause a BIR program to exceed its predefined bounds. For instance, an attempt to allocate from an exhausted collection, create more threads than allowed, or assign an integer variable a value out of its predefined range, are cases in which the program goes into a special *trap* state. This state is defined to be a self-loop state which causes the model checker to silently backtrack. The following example models an assignment of value `v` to an integer variable `x`, declared of range $MIN \ldots MAX$:

```
if
:: ! (v > MAX) ->
   if
   :: ! (v < MIN) -> x = v;
   :: else -> printf("BIR: RangeLimitException\n");
              limit_exception = true; goto endtrap;
   fi;
:: else -> printf("BIR: RangeLimitException\n");
           limit_exception = true; goto endtrap;
fi;
```

The trap state is introduced by a self-loop at the end of the thread declaration:

```
endtrap: if
           :: limit_exception -> goto endtrap;
           :: !limit_exception ->
end:          false;
         fi;
```

If the trap location is reached as result of exceeding the model bounds, the `limit_exception` flag is set and the program goes into a loop. This loop introduces a sink state into the state space of the program. Otherwise, if the trap location is reached by the normal control flow, without the `limit_exception` flag being set, the program is driven into a valid end state. The formal semantics of the bounded state-space search is given in Section 5.

4.2. The dSpin Translation

The dSPIN (*Dynamic SPIN*) model checker is designed for verification of software, providing a number of novel features on top of standard SPIN's state space reduction algorithms, e.g., partial-order reduction and state compression. Since dSPIN was originally developed as an extension of the SPIN model checker, the input language of dSPIN is a dialect of the PROMELA language [30] offering, in addition to recursive and polymorphic functions, primitives for allocating and referencing dynamic data structures. Other advantages of using dSPIN as a target model checker include the possibility of creating an unbounded number of objects and the existence of embedded on-the-fly garbage collection [36] and heap symmetry reductions [34].

Since the input of language of dSPIN is basically a superset of Promela, it is easy to modify the translation of the the previous section to target dSPIN. Currently, our translation to dSPIN takes advantage of dSPIN's dynamic object creation, garbage collection, and heap symmetry facilities. This is achieved by modifying the Spin translation of collections and dynamic allocation so that dSPIN primitives for dynamic allocation are used directly and no size bounds are associated with collections. Furthermore, when compiling Java to BIR for use with the dSPIN backend, we simply allocate one collection for each Java class, since the performance of dSPIN's native heap symmetry facilities exceeds the symmetry effect that one obtains by using a collection per object allocator.

5. Formalizing BIR

In this section we present a more detailed formalization of the semantics of BIR. To avoid excessive detail, we discuss only the formal semantics of the language constructs dealing with dynamic creation of objects and monitor-based synchronization. The interleaving semantics of a multithreaded BIR program, however, is discussed in more detail to clearly explain several variations on model semantics that our translators support.

The semantics of a BIR program is a finite transition system describing the concurrent behavior of the program as interleavings of *visible* transformations executed by threads. The operational model is layered: Section 5.0.1 defines a number of semantic domains used to describe program configurations (states). Section 5.0.2 defines the meaning of the guarded transformations that represent atomic computation steps in a BIR program. Finally, in Section 5.0.3 a transition system is defined using the semantics rules from the previous sections.

5.0.1. *Semantic Domains*

This section introduces a number of semantic domains. These domains constitute the basic layer of the operational semantics definition, capturing most of the functionality of later constructs such as expressions and actions. Conceptually, this is done by associating with every domain a number of operators, i.e., functions that manipulate values belonging to that domain. Along the presentation we draw attention to the *boundedness* of these domains. Indeed each domain is finite and operations attempting to exceed the predefined bounds will fail. This feature is important in the development of a bounded program model.

Let us first introduce some notation. For two sets A and B, $\langle a, b\rangle \in A \times B$ denotes a pair. As usual, for a relation $R \subseteq A \times B$, R^* denotes the reflexive and transitive closure. By writing s_n we denote the element found on position n in the sequence s. For a set A and a discrete element $n \in \{\bot, \epsilon\}$, let A_n denote the set $A \cup \{n\}$. Throughout this section, the notations $A_\bot$ and A_ϵ are used intensively. Intuitively, $\bot$ stands for *runtime error*, and ϵ for *undefinedness*. The *choice* operator

$c \to a \,\Box\, b$ reads "if c is true then a else b". For a mapping $m \in A \longmapsto B$ and two values $a \in A$ and $b \in B$, the mapping $[a \to b]m$ maps a to b and behaves like m for all $x \neq a$ in A. We use λ-notation for functions, where $\underline{\lambda}$ denotes *strictness* in the $\perp$ argument (passing $\perp$ as argument will cause the function to evaluate to $\perp$). Also $\lambda xy.f$ stands for $\lambda x.\lambda y.f$ and $\underline{\lambda} xy.f$ for $\underline{\lambda} x.\underline{\lambda} y.f$. For two positive integers $m < n$ we denote by $m..n$ the range $\{m, m+1, \ldots, n-1\}$. The bounded addition operator $\oplus_n$ is defined as follows: $x \oplus_n y = x + y \leq n \to x + y \,\Box\, \perp$.

Heap: As BIR is an object-based language that allows for objects to be dynamically created, there is a need for a representation of the computer's memory in our model. Nevertheless, the semantics of the memory should be abstract enough to accommodate all possible situations that can be found in practice. We represent it by means of a finite domain *Location* and an operator *nextloc*. Both are defined in Figure 10. The *nextloc* operator is not defined explicitly, we simply require it to satisfy two conditions: (i) given a location, *nextloc* will return a new available location, and (ii) the memory is exhausted after a finite number of allocations. There is a distinct location which we denote by *null*.

Figure 10 presents the definition of the dynamic memory domain, used to allocate new objects. Formally, a heap is a pair $\langle m, l \rangle$. The first (m) component of the heap is a map from memory locations to *objects*. An object is a pair $\langle s, t \rangle$ whose first component is a store and second component is an *AggregateType*. As usual, a store is a map from identifiers to values. Storing the type explicitly within the objects will allow us to quantify over all existing instances of a given type (in Section 5.0.2). The role of the second (l) component in the definition of the *Heap* is to ensure that each newly allocated object will be placed at a different location. Formally, this is guaranteed by the second (ii) property of the *nextloc* operator in the definition of the *Location* domain. The *alloc* operator takes a heap and an object as arguments. It places the object at the next available location in the heap and returns a new heap, where the l component is updated, together with the location of the newly placed object. Attempting to allocate a new object in an exhausted heap $\langle m, \perp \rangle$ will cause the *alloc* function to return $\perp$ in order to signal a runtime "out of memory" error. The *reachable* operator is used to define non-deterministic choice operators in the next section.

Collections: A BIR program does not refer directly to the heap memory, rather it uses collections to handle (bounded) dynamic object creation. Formally, a collection (Figure 11) is a pair $\langle t, i \rangle$ whose first

$$\begin{array}{rcl}
null & \in & Location \\
nextloc & : & Location \to Location_{\perp} \\
nextloc^{n}(l) & \neq & l,\ \forall\, l \in Location,\ \forall\, n \in \mathbb{N} \setminus \{0\} \quad (i) \\
nextloc^{n}(l) & = & \perp,\ \forall\, l \in Location,\ \exists\, n \in \mathbb{N} \setminus \{0\} \quad (ii) \\
 & & \\
Object & = & Store \times AggregateType \\
Heap & = & (Location \longmapsto Object_{\epsilon}) \times Location_{\perp} \\
 & & \\
alloc & : & Heap \times Object \to Heap \times Location_{\perp} \\
alloc(\langle m, l\rangle, o) & = & (\langle [l \to o]m, nextloc(l)\rangle, l)\ when\ l \neq \perp \\
alloc(\langle m, \perp\rangle, s) & = & (\langle m, \perp\rangle, \perp) \\
 & & \\
reachable & : & Heap \times Location \times Location \to \{true, false\} \\
reachable(\langle m, n\rangle, l, l') & = & \begin{cases} true & \text{if } l = l' \\ \bigvee_{\substack{i \in Identifier \\ m(l) = \langle s, t\rangle \\ s(i) = \langle k, t'\rangle}} reachable(\langle m, n\rangle, k, l') & \text{otherwise} \end{cases}
\end{array}$$

Figure 10. Heap

component specifies an aggregate type (either record or array) and second component indicates the current number of objects allocated from that collection. After a finite number of allocations, a collection is exhausted, and an attempt to allocate from an exhausted collection will result in an error. On the other hand, there is a possibility of exhausting the heap before the collection bound is exceeded. Both error situations are captured by the use of strict functions in the definition of the *new* operator. The *new* operator takes as arguments a collection and a heap. The result is a triple whose first element is an updated collection (i.e., the result of incrementing its counter), second element is an updated heap returned by an invocation to the *alloc* function, and third element is a *Reference* value (i.e., a location-type pair). Such a value is the result of applying a function (strict in both arguments) to the pair composed of the location returned by *alloc* and the integer counter obtained from the bounded increment operation $\oplus_n$. Dynamically allocated objects are referred to by *Reference* values that carry the actual type of the object along with its location in the heap memory.

Locks: Let us assume a predefined set of thread identifiers $ThreadId$. Figure 12 presents the definition of the *lock* domains. BIR locks support waiting and notification primitives. In addition, they are reentrant,

$$\begin{aligned}
Reference &= Location \times AggregateType \\
Collection_n &= AggregateType \times 0..n_\perp \\
new &: Collection_n \times Heap \to Collection_n \times Heap \times Reference_\perp \\
new(\langle t, i\rangle, h) &= (\lambda glk.(\langle t, k\rangle, g, (\underline{\lambda} l'k'.\langle l', t\rangle)(l, k)))(alloc(h, \langle zero(t), t\rangle), i \oplus_n 1)
\end{aligned}$$

Figure 11. Collection

meaning that a thread is allowed to acquire a lock multiple times, without blocking itself. Formally a lock is a 5-tuple $\langle l, t, s_w, s_n, i\rangle$ where l is the status of the lock (*free* or *taken*), t is the identifier of the thread that owns the lock (or ϵ iff the lock is free), s_w and s_n are sets of thread identifiers used for *waiting* and *notification* respectively, and i is the number of times a thread has acquired the lock. To ensure finiteness of BIR models, this number has to be bounded (by a positive integer n) as part of the definition of the lock domain ($Lock_n$). The operators associated with the lock domain in Figure 12 describe the primitive operations that involves lock objects in BIR. The *lock* and *unlock* operations acquire and release a lock object, respectively, on behalf of a given thread, given as first parameter. The reentrant nature of BIR locks is illustrated by the definition of the *lock* operator and the *lockAvailable* predicate. More precisely, a busy lock is always considered to be available to the thread that holds it. Waiting and notification on a lock object are defined by means of the *wait*, *unwait*, *notify*, and *notifyAll* operators and the *wasNotified* predicate. It is worthwhile noticing that the *wait*, *notify* and *notifyAll* functions return $\perp$ in case when the lock argument is *free*, signaling an “illegal monitor state” run-time error.

States: Program configurations (Figure 13) are represented as triplets $\langle G, H, T\rangle$ where: G is a store for global variables, H is a heap that stores dynamically allocated objects, and T is a mapping that keeps track of the local state of each thread i.e., its current control location and the values of its local variables.

In addition, we introduce two error states in order to characterize erroneous behavior in a BIR program. The first is *ErrorState* that deals with generic runtime errors, such as the failure of an expression to evaluate. The program is driven into the *LimitState* only when a bounded resource (such as the heap) has been exhausted.

$$
\begin{array}{rcl}
\mathit{LockStatus} & = & \{\mathit{free}, \mathit{taken}\} \\
\mathit{Lock}_n & = & \mathit{LockStatus} \times \mathit{ThreadId}_\epsilon \times \\
 & & \mathcal{P}(\mathit{Thread}) \times \mathcal{P}(\mathit{Thread}) \times 0..n \\
\mathit{lock} & : & \mathit{ThreadId} \times \mathit{Lock}_n \to \mathit{Lock}_{n\perp} \\
\mathit{lock}(t, \langle \mathit{free}, t', s_w, s_n, i\rangle) & = & \perp \ \mathit{when}\ t' \neq \epsilon \vee i > 0 \\
\mathit{lock}(t, \langle \mathit{free}, \epsilon, s_w, s_n, 0\rangle) & = & \langle \mathit{taken}, t, s_w, s_n, 1\rangle \\
\mathit{lock}(t, \langle \mathit{taken}, t', s_w, s_n, i\rangle) & = & (t = t') \to (\underline{\lambda} m. \langle \mathit{taken}, t, s_w, s_n, m\rangle)(i \oplus_n 1) \\
 & & \Box \ \perp \\
\mathit{unlock} & : & \mathit{ThreadId} \times \mathit{Lock}_n \to \mathit{Lock}_{n\perp} \\
\mathit{unlock}(t, \langle l, t', s_w, s_n, i\rangle) & = & \perp \ \mathit{when}\ l = \mathit{free} \vee t \neq t' \vee i = 0 \\
\mathit{unlock}(t, \langle \mathit{taken}, t, s_w, s_n, i\rangle) & = & (i = 1) \to \langle \mathit{free}, \epsilon, s_w, s_n, 0\rangle \\
 & & \Box\ \langle \mathit{taken}, t, s_w, s_n, i - 1\rangle\ \mathit{when}\ i > 0 \\
\mathit{lockAvailable} & : & \mathit{ThreadId} \times \mathit{Lock}_n \to \mathit{Boolean} \\
\mathit{lockAvailable}(t, \langle l, t', s_w, s_n, i\rangle) & = & (l = \mathit{free}) \vee (t = t') \\
\mathit{hasLock} & : & \mathit{ThreadId} \times \mathit{Lock}_n \to \mathit{Boolean} \\
\mathit{hasLock}(t, \langle l, t', s_w, s_n, i\rangle) & = & (l = \mathit{taken}) \wedge (t = t') \\
\mathit{wait} & : & \mathit{ThreadId} \times \mathit{Lock}_n \to (0..n \times \mathit{Lock}_n)_\perp \\
\mathit{wait}(t, \langle l, t', s_w, s_n, i\rangle) & = & \perp \ \mathit{when}\ l = \mathit{free} \vee t \neq t' \vee i = 0 \\
\mathit{wait}(t, \langle \mathit{taken}, t, s_w, s_n, i\rangle) & = & (i, \langle \mathit{free}, \epsilon, s_w \cup \{t\}, s_n, 0\rangle)\ \mathit{when}\ i > 0 \\
\mathit{unwait} & : & \mathit{ThreadId} \times 0..n \times \mathit{Lock}_n \to \mathit{Lock}_{n\perp} \\
\mathit{unwait}(t, i, \langle l, t', s_w, s_n, i'\rangle) & = & \perp \ \mathit{when}\ l = \mathit{taken} \vee t' \neq \epsilon \vee t \notin s_n \vee i' > 0 \\
\mathit{unwait}(t, i, \langle \mathit{free}, \epsilon, s_w, s_n, 0\rangle) & = & \langle \mathit{taken}, t, s_w, s_n \setminus \{t\}, i\rangle \\
\mathit{notify} & : & \mathit{ThreadId} \times \mathit{Lock}_n \to \mathit{Lock}_{n\perp} \\
\mathit{notify}(t, \langle l, t', s_w, s_n\rangle) & = & \perp \ \mathit{when}\ l = \mathit{free} \vee t \neq t' \\
\mathit{notify}(t, \langle \mathit{taken}, t, \emptyset, s_n\rangle) & = & \langle \mathit{taken}, t, \emptyset, s_n\rangle \\
\mathit{notify}(t, \langle \mathit{taken}, t, s_w, s_n\rangle) & = & \langle \mathit{taken}, t, s_w \setminus \{t'\}, s_n \cup \{t'\}\rangle\ \mathit{when}\ \exists t' \in s \\
\mathit{notifyAll} & : & \mathit{ThreadId} \times \mathit{Lock}_n \to \mathit{Lock}_{n\perp} \\
\mathit{notifyAll}(t, \langle l, t', s_w, s_n\rangle) & = & \perp \ \mathit{when}\ l = \mathit{free} \vee t \neq t' \\
\mathit{notifyAll}(t, \langle \mathit{taken}, t, \emptyset, s_n\rangle) & = & \langle \mathit{taken}, t, \emptyset, s_n\rangle \\
\mathit{notifyAll}(t, \langle \mathit{taken}, t, s_w, s_n\rangle) & = & \langle \mathit{taken}, t, \emptyset, s_w \cup s_n\rangle\ \mathit{when}\ s_w \neq \emptyset \\
\mathit{wasNotified} & : & \mathit{ThreadId} \times \mathit{Lock}_n \to \mathit{Boolean} \\
\mathit{wasNotified}(t, \langle l, t', s_w, s_n\rangle) & = & (t \in s_n)
\end{array}
$$

Figure 12. Lock Domains

$$\begin{aligned} \widehat{State} &= Global \times Heap \times ThreadPool \\ State &= \widehat{State} \cup \{ErrorState, LimitState\} \end{aligned}$$

Figure 13. Program States

5.0.2. *Transformations*

The executable part of a BIR thread is a finite sequence of guarded transformations. In this section we define the semantics of guarded transformations. In order to do so, we consider as predefined a number of semantic judgments. Namely, the $\langle G, H, T\rangle \vdash^t_{expr}$ `ast` $\leadsto val$ operator maps an abstract syntax tree fragment `ast` to a value val that represents its value in the program state $\langle G, H, T\rangle$. The evaluation of expression `ast` is carried out by the thread denoted by $t \in ThreadId$. As usual, the notation $\langle G, H, T\rangle \vdash^t_{expr}$ `ast` $\leadsto \perp$ denotes failure of `ast` to evaluate in state $\langle G, H, T\rangle$. The semantics of actions is captured by the derivation operator $\langle G_i, H_i, T_i\rangle \vdash^t_{act}$ `ast` $\leadsto \langle G_j, H_j, T_j\rangle$ which describes the transformation of a program state $\langle G_i, H_i, T_i\rangle$ under the action represented by the abstract syntax tree fragment `ast`, the resulting state being $\langle G_j, H_j, T_j\rangle$. For the assignment actions we consider a new judgment $\langle G_i, H_i, T_i\rangle \vdash^v_{asgn}$ `lhs` $\leadsto \langle G_j, H_j, T_j\rangle$ describing the effect of assigning an explicit value v to the left-hand side expression `lhs` in state $\langle G, H, T\rangle$.

Being a concurrent asynchronous system, a BIR program is inherently non-deterministic. However, in addition to the non-determinism caused by the parallel composition of threads, the language allows for non-determinism even in a sequential context, namely inside a thread. This is a powerful language tool for describing systems allowing abstraction by conservative over-approximation of concrete behaviors [16]. The intuition is that a model checking tool will exhaustively explore all possible states that result from an application of a choice rule. The rules defining the semantics of non-deterministic choices are given in Figure 14. Each choice rule defines a judgment of the form $\langle G_i, H_i, T_i\rangle \vdash^t_{choice}$ `ast` $\leadsto \langle G_j, H_j, T_j\rangle$ that describes the effect of the non-deterministic assignment. These rules can be applied non-deterministically because of the existential quantifier that appears in the precondition of each rule.

The first rule in Figure 14 defines the meaning of a choice between several given values in assignment. One of the offered expressions is chosen non-deterministically and assigned to the left-hand side of the choice action, the result of this assignment being the result of the choice action. This rule handles also the error scenario in which the expression fails to evaluate, case in which S is either $ErrorState$ or $LimitState$.

The distinction between the semantics of the `internChoose` and `externChoose` (Rule (2)) actions will become more clear in Section 5.0.3. To give the intuition behind this, let us assume that the BIR program is obtained from an open module for which an environment has been previously synthesized. Both the module and the environment can perform non-deterministic actions, however only the module's (internal) non-deterministic actions can be the result of an abstraction of the original system, and therefore may generate a spurious counterexample when model-checked. To avoid the report of such counterexamples, one approach is to cut the exploration of the current path whenever an `internChoose` action is encountered. Formally, we distinguish an `internChoose` which is defined by a $s \vdash^t_{choice}$ `ast` $\rightsquigarrow s'$ judgement from an `externChoose` which is described by a $s \vdash^t_{act}$ `ast` $\rightsquigarrow s'$ judgement.

In languages with dynamic creation of objects the "static" form of non-deterministic choice is not sufficient. Indeed, some properties have to be verified with respect to each instance of a given class. To model non-deterministic choices among instances of a class, we introduce the `reachable` and `forall` actions. Rule (2) captures the semantics of a choice over all instances of a given type that are `reachable` starting with a given location. The reachability information is captured by the *reachable* predicate defined in Figure 10. Rule (3) defines the choice over all existing instances of a given type. Both rules are applicable if there exists at least a (reachable) location l in the heap that refers to an instance of the given type or any of its subtypes. The semantics of a non-deterministic choice over all (or reachable) instances in case there are no such instances is given by the rules (4) and (5): in these cases the choice action does not change the program state.

Formally, we define a *transformation* relation $\mapsto \in State \times ThreadId \times State$. We use the notation $s \overset{t}{\mapsto} s'$ for $\langle s, t, s' \rangle \in \mapsto$. We define two successor and two predecessor functions as follows:

$$\begin{array}{rlrl} out(s,t) = \{s' \mid s \overset{t}{\mapsto} s'\} & & in(s,t) = \{s' \mid s' \overset{t}{\mapsto} s\} \\ Out(s) = \bigcup_t out(s,t) & & In(s) = \bigcup_t in(s,t) \end{array}$$

We explicitly denote transformations whose results involve non-deterministic choice. We do so with a separate transformation relation $\mapsto^{\prec} \in State \times ThreadId \times State$ that reflects only the result of executing a choice action at some point during the transformation. Recall that, in BIR, transformations are sequences of atomic actions, so performing one `choose` action will cause the entire transformation to be non-deterministic.

There are two kinds of transformations in BIR: *visible* and *invisible*. Intuitively, the effect of an invisible transformation should not be

$$\frac{\exists\, 1 \le i \le n\ [\langle G,H,T\rangle \vdash^t_{act} \texttt{lhs} := \texttt{e}_\texttt{i} \rightsquigarrow S]}{\langle G,H,T\rangle \vdash^t_{choice} \texttt{lhs} := \texttt{internChoose}(\texttt{e}_1,\ldots,\texttt{e}_\texttt{n}) \rightsquigarrow S} \quad (1)$$

$$\frac{\exists\, 1 \le i \le n\ [\langle G,H,T\rangle \vdash^t_{act} \texttt{lhs} := \texttt{e}_\texttt{i} \rightsquigarrow S]}{\langle G,H,T\rangle \vdash^t_{act} \texttt{lhs} := \texttt{externChoose}(\texttt{e}_1,\ldots,\texttt{e}_\texttt{n}) \rightsquigarrow S}$$

$$\frac{\exists\, l \begin{bmatrix} \langle G,H,T\rangle \vdash^t_{expr} \texttt{lhs}_2 \rightsquigarrow \langle l',y'\rangle & & \mathit{reachable}(H,l,l') \\ H = \langle m,n\rangle & m(l) = \langle s,y\rangle & \langle y,id\rangle \in \mathit{SubType}^* \\ & \langle G,H,T\rangle \vdash^l_{asgn} \texttt{lhs}_1 \rightsquigarrow S & \end{bmatrix}}{\langle G,H,T\rangle \vdash^t_{choice} \texttt{lhs}_1 := \texttt{reachable}(\texttt{lhs}_2,\texttt{id}) \rightsquigarrow S} \quad (2)$$

$$\frac{\exists\, l \begin{bmatrix} H = \langle m,n\rangle \ \ m(l) = \langle s,y\rangle \ \ \langle y,id\rangle \in \mathit{SubType}^* \\ \langle G,H,T\rangle \vdash^l_{asgn} \texttt{lhs} \rightsquigarrow S \end{bmatrix}}{\langle G,H,T\rangle \vdash^t_{choice} \texttt{lhs} := \texttt{forall}(\texttt{id}) \rightsquigarrow S} \quad (3)$$

$$\frac{\nexists\, l \begin{bmatrix} \langle G,H,T\rangle \vdash^t_{expr} \texttt{lhs}_2 \rightsquigarrow \langle l',y'\rangle & & \mathit{reachable}(H,l,l') \\ H = \langle m,n\rangle & m(l) = \langle s,y\rangle & \langle y,id\rangle \in \mathit{SubType}^* \end{bmatrix}}{\langle G,H,T\rangle \vdash^t_{choice} \texttt{lhs}_1 := \texttt{reachable}(\texttt{lhs}_2,\texttt{id}) \rightsquigarrow \langle G,H,T\rangle} \quad (4)$$

$$\frac{\nexists\, l\, [\, H = \langle m,n\rangle \ \ m(l) = \langle s,y\rangle \ \ \langle y,id\rangle \in \mathit{SubType}^*\,]}{\langle G,H,T\rangle \vdash^t_{choice} \texttt{lhs} := \texttt{forall}(\texttt{id}) \rightsquigarrow \langle G,H,T\rangle} \quad (5)$$

Figure 14. Choices

observed by threads other than the one containing the transformation. The BIR generator will have to perform static checks that conservatively identify invisible transformations. For instance, a transformation that only writes into local variables can be safely labeled as invisible[2]. Notice that it is conservative to label an invisible transformation as visible, whereas the converse does not hold. Formally, the $\mapsto$ relation is partitioned into a *visible* relation $\mapsto_{vis}$ and an *invisible* one $\mapsto_{inv}$.

We now define the visible transformation between states $\mapsto_{vis}$, as the least relation that meets rules (6, 8, 9) in Figure 15. The *Code* function is used to map a syntactic location into the set of statements it labels. Rule (6) defines the meaning of a successful transformation. As usual, the transformation can occur if the control location of the acting thread matches the source location of the transformation and the guard evaluates to *true*. The transformation succeeds if and only if all atomic actions from the transformation's body can succeed. On the other hand, rules (8) and (9) deal with errors. Namely, a transformation fails if either the guard fails to evaluate or one action fails to complete (drives the program into an error state). The invisible transformation relation $\mapsto_{inv}$ is the least relation that meets rule (7). We use here the syntax

[2] For more detail, we refer the interested reader to [23] for a formal definition of action independence and to [31] for a definition of action safety.

for the `invisible` guarded transformations (Appendix A). Notice that the error states can only be reached by visible transformations.

To ensure the correct partitioning of the transformation relation $\mapsto$, we must impose two syntactic restrictions on the syntax of a BIR thread: (1) it is illegal to have a visible and an invisible transformation originating from and ending at the same location, and, (2) it is illegal to have an invisible transformation originating from and ending at the same location. It can be proven that these restrictions are sufficient to ensure the distinction between the visible and invisible transformation relations (Proposition 1 in Appendix B), and formally we have $\mapsto_{vis} \cap \mapsto_{inv} = \emptyset$. The transformation relation $\mapsto$ is then defined as: $\mapsto = \mapsto_{vis} \cup \mapsto_{inv}$.

To define the non-deterministic version of the transformation relation ($\mapsto^{\prec}$) we use a similar reasoning as in the case of $\mapsto$. The only difference is that at least one of the actions of the guarded transformation has to be interpreted (in the preconditions of the defining rules) using a choice judgement of the form $s \vdash^{t}_{choice} \mathsf{a} \rightsquigarrow s'$. To enforce the semantic distinction between deterministic and choice transformations ($\mapsto \cap \mapsto^{\prec} = \emptyset$), we impose a sufficient syntactic restriction that is similar to the one concerning visibility of transformations: it is illegal for two or more transformations, only one of which containing choose actions, to begin from and end at the same location.

5.0.3. *Transition System*

We are now ready to describe the execution of a BIR program by a labeled transition system $M = (\Sigma, S, T)$ where S is a set of states and $\longrightarrow \subseteq S \times \Sigma \times S$ is a labeled transition relation between states. There are four labels in the alphabet, each of them being a pair: $\Sigma = \{vis, inv\} \times \{\prec, -\}$ where, for any two states $s, s' \in S$ and for some thread $t \in ThreadId$:

1. $\langle s, (vis, *), s' \rangle \in \longrightarrow$ $(s \longrightarrow^{*}{}_{vis}\, s')$, if $s \overset{t}{\mapsto}_{vis} s'$ or $s \overset{t}{\mapsto}{}^{\prec}{}_{vis}\, s'$.

2. $\langle s, (inv, *) s' \rangle \in \longrightarrow$ $(s \longrightarrow^{*}{}_{inv}\, s')$, if $s \overset{t}{\mapsto}_{inv} s'$ or $s \overset{t}{\mapsto}{}^{\prec}{}_{inv}\, s'$.

3. $\langle s, (*, \prec), s' \rangle \in \longrightarrow$ $(s \longrightarrow_{*}{}^{\prec} s')$, if $s \overset{t}{\mapsto}{}^{\prec}{}_{vis}\, s'$ or $s \overset{t}{\mapsto}{}^{\prec}{}_{inv}\, s'$.

4. $\langle s, (*, -), s' \rangle \in \longrightarrow$ $(s \longrightarrow_{*} s')$, if $s \overset{t}{\mapsto}_{vis} s'$ or $s \overset{t}{\mapsto}_{inv} s'$.

We shall first define a basic transition system M, the result of a classical state-space exploration algorithm used in explicit-state model checking [30]. We then formally define the *bounded* version of M, denoted by M_B, which is the result of resuming the state-space search

$$\frac{\begin{array}{c}\langle \texttt{when (e) do \{a}_1,\ldots,\texttt{a}_n\texttt{\} goto m}\rangle \in \mathit{Code}(l)\\ T(t) = \langle l, n, \mathit{active}, \sigma\rangle \quad \langle G,H,T\rangle \vdash^t_{expr} \texttt{e} \rightsquigarrow \mathit{true}\\ \langle G,H,T\rangle \vdash^t_{act} \texttt{a}_1 \rightsquigarrow \langle G_1,H_1,T_1\rangle\\ \ldots\\ \langle G_{n-1},H_{n-1},T_{n-1}\rangle \vdash^t_{act} \texttt{a}_n \rightsquigarrow \langle G_n,H_n,T_n\rangle\\ T_n(t) = \langle l, n', s', \sigma'\rangle \quad T' = [t \to \langle m, n', s', \sigma'\rangle]T_n\end{array}}{\langle G,H,T\rangle \overset{t}{\mapsto}_{vis} \langle G_n,H_n,T'\rangle} \quad (6)$$

$$\frac{\begin{array}{c}\langle \texttt{when (e) do invisible \{a}_1,\ldots,\texttt{a}_n\texttt{\} goto m}\rangle \in \mathit{Code}(l)\\ T(t) = \langle l, n, \mathit{active}, \sigma\rangle \quad \langle G,H,T\rangle \vdash^t_{expr} \texttt{e} \rightsquigarrow \mathit{true}\\ \langle G,H,T\rangle \vdash^t_{act} \texttt{a}_1 \rightsquigarrow \langle G_1,H_1,T_1\rangle\\ \ldots\\ \langle G_{n-1},H_{n-1},T_{n-1}\rangle \vdash^t_{act} \texttt{a}_n \rightsquigarrow \langle G_n,H_n,T_n\rangle\\ T_n(t) = \langle l, n', s', \sigma'\rangle \quad T' = [t \to \langle m, n', s', \sigma'\rangle]T_n\end{array}}{\langle G,H,T\rangle \overset{t}{\mapsto}_{inv} \langle G_n,H_n,T'\rangle} \quad (7)$$

$$\frac{\begin{array}{c}\langle \texttt{when (e) do [invisible] \{a}_1,\ldots,\texttt{a}_n\texttt{\} goto m}\rangle \in \mathit{Code}(l)\\ T(t) = \langle l, n, \mathit{active}, \sigma\rangle \quad \langle G,H,T\rangle \vdash^t_{expr} \texttt{e} \rightsquigarrow \bot\end{array}}{\langle G,H,T\rangle \overset{t}{\mapsto}_{vis} \mathit{ErrorState}} \quad (8)$$

$$\frac{\begin{array}{c}\langle \texttt{when (e) do [invisible] \{a}_1,\ldots,\texttt{a}_n\texttt{\} goto m}\rangle \in \mathit{Code}(l)\\ T(t) = \langle l, n, \mathit{active}, \sigma\rangle \quad \langle G,H,T\rangle \vdash^t_{expr} \texttt{e} \rightsquigarrow \mathit{true}\\ \langle G,H,T\rangle = \langle G_0,H_0,T_0\rangle \vdash^t_{act} \texttt{a}_1 \rightsquigarrow \langle G_1,H_1,T_1\rangle\\ \ldots\\ \langle G_{i-1},H_{i-1},T_{i-1}\rangle \vdash^t_{act} \texttt{a}_i \rightsquigarrow S \quad 1 \le i \le n\\ S \ \in \ \{\mathit{LimitState}, \mathit{ErrorState}\}\end{array}}{\langle G,H,T\rangle \overset{t}{\mapsto}_{vis} S} \quad (9)$$

Figure 15. Guarded Transformations

whenever a resource bound was exceeded. Next, we define the deterministic (or *choose-free*) version of M, denoted by M_D, which is the result of a state-space exploration aimed at producing guaranteed feasible counter-examples of safety properties [46]. The explicit labeling of non-deterministic transitions is needed in order to define the choose-free version of the transition system. The visible/invisible labeling of transitions is meaningful in order to define further optimizations of the state-space, such as *virtual coarsening*.

In the basic version of the transition system (M), a sequence of invisible transformations performed by the same thread cannot be interleaved with transformations of different threads. Notice that this is a safe assumption, since an action is declared invisible, assuming that it is *globally independent* [31] with respect to all other transformations of other threads. An invisible action is however not independent with other non-deterministic choices of the same thread, and for this reason we need to preserve the internal branching structure of a thread when

$$\frac{s \overset{t}{\mapsto}_{inv} s' \quad out(s',t) \neq \emptyset}{\langle s,\epsilon\rangle \hookrightarrow_{inv} \langle s',t\rangle \quad \langle s,t\rangle \hookrightarrow_{inv} \langle s',t\rangle} \tag{10}$$

$$\frac{s \overset{t}{\mapsto}_{vis} s'}{\langle s,t\rangle \hookrightarrow_{vis} \langle s',\epsilon\rangle \quad \langle s,\epsilon\rangle \hookrightarrow_{vis} \langle s',\epsilon\rangle} \tag{11}$$

$$\frac{s \overset{t'}{\mapsto}_{inv} s' \quad out(s',t) = \emptyset}{\langle s,\epsilon\rangle \hookrightarrow_{vis} \langle s',\epsilon\rangle \quad \langle s,t\rangle \hookrightarrow_{vis} \langle s',\epsilon\rangle} \tag{12}$$

Figure 16. Pseudo-transition System

defining the transition system. It can been shown that treating invisible transformations by disallowing interleavings with other threads generates a labeled transition system that is branching bisimilar [20] to the fully interleaved one. Assuming that the invisible labeling of transformations preserves state stuttering i.e., two states connected by an invisible transition will satisfy the same set of predicates, this semantics strongly preserves the truth value of formulas written in the *next-free* CTL* temporal logic [44].

We describe the interleaving semantics of invisible actions by defining a system of *pseudo-transitions* $\mathcal{M} = (\Sigma, \mathcal{S}, \hookrightarrow)$, where:

- $\mathcal{S} = State \times ThreadId_\epsilon$ and,
- $\hookrightarrow \in \mathcal{S} \times \Sigma \times \mathcal{S}$ is the least relation defined by the rules in Figure 16. Note that we only give rules for the deterministic pseudo-transitions here; the non-deterministic choice pseudo-transitions $\hookrightarrow^{\prec}$ can be derived analogously, using the choice transformation relation $\mapsto^{\prec}$.

Intuitively, rule (10) defines the beginning and the span of a sequence of invisible pseudo-transitions. The sequence begins with a pair $\langle s, \epsilon\rangle$ when an invisible transformation is performed by a thread t. As result, the next pair remembers the acting thread together with the successor state $\langle s', t\rangle$. The sequence can be continued as long as there exists an invisible transformation that can be performed by t, as described by the right-hand side of the conclusion. Notice that invisible pseudo-transitions can occur as long as the acting thread t is not blocked in the destination state s' of the transformation, a condition that is expressed formally by the requirement $out(s',t) \neq \emptyset$. In the same style of reasoning, rule (11) defines visible pseudo-transitions. A visible pseudo-transition is the result of a visible transformation performed by a thread t. The left hand side of the conclusion describes the situation when a visible pseudo-transition ends a sequence of invisible transitions by resetting the thread identifier to ϵ in the successor state-thread pair. The right hand side of rule's (11) conclusion specifies a default visible pseudo-transition between two pairs. Finally, rule (12) describes the end of an invisible sequence in the case when the acting thread t is blocked in the destination state ($out(s', t) = \emptyset$). In this case, the invisible transformation $s \overset{t}{\mapsto}_{inv} s'$ gives rise to a visible pseudo-transition that ends the invisible sequence.

We can now define the basic transition system, that will be generated by a classical state-space search $M = (\Sigma, S, \longrightarrow)$:

- $S = State$ and,
- $$\longrightarrow = \bigcup \begin{array}{c} \{\langle s, (x,y), s' \rangle \mid \exists u, v \in ThreadId_\epsilon \ . \ \langle s, u \rangle \hookrightarrow_x^y \langle s', v \rangle\} \\ \{\langle ErrorState, (vis, -), ErrorState \rangle\} \\ \{\langle LimitState, (vis, -), LimitState \rangle\} \end{array}$$

The set of states is the set of program configurations, as defined in Figure 13. There exists a transition between two states whenever there exists a pseudo-transition between two state-thread pairs (ignoring the thread component) and the transition labeling (x, y) is the same for a transition as for a pseudo-transition between same states. It can be proven that the transition labeling is indeed well-defined (Proposition 2 in Appendix B). Moreover, we add two visible deterministic transitions to our model, making *ErrorState* and *LimitState* become sink states, as a consequence of the fact that the explicitly added transitions are the only possible outgoing transitions from an error state.

Bounded Transition System: There are situations in which one is interested in verifying a property only on sequences of states that respect certain constraints. This feature can be obtained in BIR by setting explicit bounds on each resource of the program, such as the maximum number of objects that can be allocated by an allocator or the maximum number of threads that can be dynamically created by a `start` action. Exceeding these bounds in a classical search will drive the system into the *LimitState*. This is usually an observable move of the system and, in practice, reaching *LimitState* will stop the classical state-space search, issuing an error message. In the bounded state-space search, the possibility of exceeding a bound is detected in advance and the search backtracks from the current state, ignoring the limit error. Formally, this style of search is defined by a bounded transition system $M_B = (\Sigma, S, \longrightarrow_B)$ that is derived from the basic version $M = (\Sigma, S, \longrightarrow)$ as follows:

$$\begin{aligned} \longrightarrow_B = \longrightarrow \ \setminus \ & \{\langle s, (*, *), LimitState \rangle \mid s \in In(LimitState)\} \\ \cup \ & \{\langle s, (vis, -), s \rangle \mid s \in In(LimitState)\} \end{aligned}$$

That is, we eliminate all transitions that lead to *LimitState* and add instead visible self loops to each of the states preceding *LimitState* in the original transition system.

Choose-free Transition System: The choose-free search [46] aims at finding only feasible counterexample of temporal logic properties. A counterexample $w = s_0 \longrightarrow s_1 \longrightarrow \ldots$ is an infinite sequence of states and transitions which does not satisfy a given temporal logic formula. For instance, a counterexample for an LTL formula ϕ or a ACTL formula $A\phi$ is a path w in the transition system such that $w \not\models \phi$. [3]

In practice, a counterexample is said to be *feasible* if it corresponds to a realistic computation of the original system i.e., before abstraction is performed. However deciding whether a counterexample is actually feasible can be expensive. An alternative is to trade soundness of the model checking procedure for the guarantee that every counterexample found is a real one. In this particular setting, a counterexample is said to be *infeasible* if it contains at least one transition that occurs as the result of a non-deterministic choice in the BIR program. Notice that this is a conservative definition. A state-space search is *choose-free* if it avoids taking non-deterministic transitions. We formally specify such a state-space search by defining a deterministic transition system $M_D = (\Sigma, S, \longrightarrow_D)$ derived from the basic one $M = (\Sigma, S, \longrightarrow)$ as follows:

$$\longrightarrow_D = \longrightarrow \ \setminus \{\langle s, (*, \prec), s' \rangle\}$$

Since choice transitions $s \longrightarrow^{\prec} s'$ are labeled according to their non-deterministic origin (Figure 14), we simply exclude them from the original transition relation. Any path in the deterministic model will not contain choice transitions, therefore any counterexample found by model checking on M_D will be feasible.

Virtual Coarsening: The notion of virtual coarsening stems from the pioneering work of Ashcroft and Manna [2], and later Pnueli [47], in using automatic deduction to prove correctness properties of concurrent programs. The idea is to optimize a parallel program for verification, by lumping together computation steps that are guaranteed to perform only transformations that are local to a process. Since in a correctly generated BIR program this condition should be met by any invisible transformation, coarsening a BIR thread amounts to grouping together sequences of invisible transformations. Figure 17 shows how a sequence

[3] For ECTL formulas, the model checker can only issue paths that testify for the correctness of formulas.

```
loc l1_a: when e1 do invisible { a1; ... an; } goto l2;
loc l2_a: when e2 do { b1; ... bm; } goto l3;

loc l1_b:
   when e1 && e2 do { a1; ... an; b1; ... bm; } goto l3;
   when e1 && !e2 do { a1; ... an; } goto l2;
loc l2_b: when e2 do { b1; ... bm; } goto l3;
```

Figure 17. Virtual Coarsening in BIR

composed of an invisible and a visible transformation (denoted with `_a`) can be coarsened. The resulting transitions (denoted with `_b`) consist of a new transformation whose guard is the conjunction of the guards from the original transformations `e1` and `e2` and whose body is the concatenation of the bodies belonging to the original transformation. Whenever it is possible (i.e., when `e1 && e2` is true), the newly introduced transformation is executed, and the intermediate state, in which control is at location `l2`, is skipped. However if the newly introduced transformation cannot be executed, the original computation is performed. Notice that the two transformations originating at location `l1_b` are deterministic, due to the `!e2` conjunct inserted in the guard of the second one.

Since a thread is allowed to non-deterministically choose between invisible transformations, the coarse transitions should not "cross" branching points within a thread declaration. Otherwise, the branching structure of the state-space will be lost due to virtual coarsening, and therefore the truth value of formulas of a branching-time temporal logic, such as the next-free fragment of CTL [8] might not be preserved.

Before proceeding with the description of virtual coarsening, we draw attention upon the following issue: since sequences of invisible transformations will be performed without interleaving, there is a one-to-one correspondence between a sequence of invisible *transformations* performed by a thread and the invisible *transitions* it generates. It is therefore correct to work directly with transition systems in defining the reduction.

We will formalize virtual coarsening directly on the transition system $M = (\Sigma, S, \longrightarrow)$, and prove its correctness using branching bisimulation equivalence [22] between transition systems. Let $R \subseteq S \times S$ be a relation on states defined as follows:

$$R = \{\langle s, s\rangle \mid s \in S\} \cup \{\langle s, s'\rangle, \langle s', s\rangle \mid s \Rightarrow s'\}$$

where $s \Rightarrow s'$ if and only if there exists a finite path $s_0 \longrightarrow_{inv} s_1 \longrightarrow_{inv} \dots s_{n-1} \longrightarrow_{inv} s_n$, for some $n \geq 1$, such that:

i) $s_0 = s$ and $s_n = s'$,

ii) $\forall\, 0 \leq i < n \,.\, Out(s_i) = \{s_{i+1}\}$,

iii) $\forall\, 0 < i \leq n \,.\, In(s_i) = \{s_{i-1}\}$.

The intuition behind the definition of $\Rightarrow$ is the following: we are allowed to lump together as many invisible actions as possible, given that they all belong to a sequential path (ii) and there are no other incoming transitions to the states on the path (iii). Since the goal of the reduction is to eliminate the intermediate states $s_1, \dots, s_{n-1}, s_n$, the latter condition is needed to preserve those states that are destinations for transitions other than the ones belonging to the path.

In fact R is an equivalence relation; by definition it is reflexive and symmetric, and transitivity follows immediately from the definition of $\Rightarrow$. Let $[s]_R$ denote the equivalence class of a state s with respect to R. The coarse transition system is defined as the quotient of M with respect to R. The states of a quotient system are equivalence classes of states from the original system, namely $M_{/R} = (\Sigma, S_R, \longrightarrow_R)$ where:

– $S_R = \{[s]_R \mid s \in S\}$ and,

– $\longrightarrow_R = \{\langle [s]_R, [s']_R \rangle \mid s \longrightarrow s'\}$.

For the purposes of explicit-state model checking we represent the quotient system by a projection $h : S_R \to S$ where $h([s]_R) = s'$ such that $s' \in [s]_R$ and for no $s'' \in [s]_R$ we have $s'' \Rightarrow s'$. Intuitively, a representative of an equivalence class is the first state that can be reached by a depth-first search. It follows (Proposition 3 in Appendix B) that such a state is unique and therefore h is a well-defined function. Moreover, it is also injective. Formally, the explicit-state reduced transition system is $M_h = (\Sigma, h(S_R), \longrightarrow_h)$ where $s \longrightarrow_h s'$ if and only if $h^{-1}(s) \longrightarrow_R h^{-1}(s')$. Obviously, h is an isomorphism between $M_{/R}$ and M_h. The correctness of our construction is ensured by the fact that M and M_h are branching bisimilar (Proposition 4 in Appendix B). Together with the assumption that the invisible labeling of transitions and stuttering with respect to a set of predicates are consistent, it follows that all temporal logic formulas written in CTL*_X are preserved by virtual coarsening [44].

Virtual coarsening is implemented in Bandera in as a syntactic transformation at the BIR level. Since a representative state is the first reachable state in the equivalence class, a transition between two states corresponds to a maximal deterministic sequence of invisible transitions ending either with a non-deterministic invisible transition or a visible transition. Simple conservative tests can be done to ensure that an invisible transformation is deterministic, in order to deal with the first reduction rule. The latter case is exemplified by the syntactic coarsening in Figure 17. Notice that the syntactic transformations are only approximative; for instance, inferring that two transformation guards are actually disjoint can be an undecidable problem. In such cases, the control flow graph of a thread can be used as a conservative approximation of its actual behavior.

6. Related Work

There are several noteworthy projects on software model-checking. Since this paper focuses on using intermediate language to stage translations to model-checking engines, our discussion of related work will focus on tool environments with similar goals.

The design goals of the IF2.0 validation environment [41], developed at Verimag, are similar to those of the Bandera project in that both rely on intermediate forms to aid in the translation of design notations to model-checking tools. Specifically, IF relies on a dedicated intermediate format to translate from high-level specification formalisms such as SDL or UML state machines into a description of communicating state machines. The specification language describes a set of *dynamically created* processes connected via asynchronous buffers and shared variables. Real-time modeling is supported, as each process may use several clocks to measure time during execution and transformations may be guarded time constraints. The IF type system provides complex data types, such as enumeration, range, array and record. The IF language is understood by a number of validation tools, such as static analyzers (LIVE) and translators towards labeled transition systems (LTS) and PROMELA (IF2PML). The former are used to reduce the size of the models, while the latter open the possibilities for model checking and test generation.

Some primary differences between IF and BIR are that IF includes various features omitted from BIR such as clocks and event buffers. Both of these could be useful additions to BIR, for example, clocks might provide the basis for checking timing-related properties of the Embedded Java and Real-time Java dialects. On the other hand, BIR provides features omitted from IF to model Java software including locks and dynamic object creation. There are currently no translations from Java or other high-level programming languages to IF.

SAL (Symbolic Analysis Laboratory) is a framework for synergistically combining model-checking, theorem-proving, and static analysis tools for verification of concurrent systems. The heart of SAL is the SAL intermediate language developed by SRI in collaboration with groups at Stanford, Berkeley, and Verimag for specifying concurrent systems in a compositional way [5]. The datatypes of SAL are very similar to those of IF. SAL provides both synchronous and asynchronous composition of modules. Translations from SAL to PVS [45] and SMV [6] have been implemented, and other tools for predicate abstraction, invariant generation, and slicing have been integrated. Unlike BIR, SAL is not specifically targetted to modeling dynamic aspects of software systems, consequently, translations from languages like Java are more complex.

The Java Path Finder model-checker [54] works directly on Java bytecode. There are several advantages to having a verification tool work directly on Java bytecode as opposed to working on an alternate notation such as BIR. The semantics of Java bytecode is already well-defined. Moreover, Java bytecode is widely used, the Java to bytecode translation is well-understood and widely implemented, and there are numerous tools that also work directly on bytecodes. The down side of having tools work on bytecode directly is that it can be difficult to customize the model based on how the program uses a particular feature. For example, if an object's lock is used but wait and notify are not, then a direct interpretation approach like JPF will still maintain a representation of the wait-set which is a waste of space.

Gerard Holzmann's FeaVer tool also translates a general-purpose programming language (i.e., C) into Promela for checking with SPIN [33]. Feaver does not use an intermediate language but instead relies to some degree on the syntactic similarities between C and Promela. Specifically, it uses a pattern-matching approach where an application-specific lookup table associates C code patterns or fragments with corresponding Promela fragments. Translation from C to Promela proceeds by traversing the C program and applying the mappings from the table to individual C fragments to obtain Promela fragments. While Holzmann has demonstrated that this approach can be very effective in checking large telephony applications, it does not seem amenable to providing a robust interoperability platform between other input notations or model-checking tools other than SPIN. The recent release of SPIN version 4.0 adds the capability of embedding fragments of C code into Promela. This allows statically named C variables to be added to the state vector and C expressions and statements to be used to implement state transitions; this can provide an effective means for reusing C code to implement complex state transitions. It does not, however, support the kind of dynamic data structures that characterize modern software; heap-allocated C structures cannot be stored in the state vector. BIR collections directly support dynamic data and the forall and reachable non-deterministic choice constructs allow those structures to be queried effectively.

Finally, other work on software model-checking make unique and interesting contributions such as the SLAM tool [4] from Microsoft Research which implements an automated predicate abstraction methodology for sequential C programs, Godefroid's Verisoft tool for stateless checking of concurrent C systems [24], Stoller's [49] tool for stateless checking of multi-threaded distributed Java programs, and Yahav's work on checking safety properties of Java programs [55] built on top of Lev-Ami and Sagiv's three-valued logic analysis tool (TVLA) [38]. Since these tools do not make use of an intermediate language as an exchange format for communicating with multiple back-ends, we do not give a deeper assessment of these here.

7. Conclusion

The goal of this paper has been to provide a comprehensive account of the Bandera Intermediate Representation including our design goals, BIR's syntax and semantics, and strategies for translating to and from BIR. The design of BIR has proven effective in supporting model-checking properties of a variety of real concurrent Java applications and other software design notations.

We believe that model checker input languages should evolve to support the needs of emerging applications of model checking as a software analysis technology. We believe that experience with BIR can help shape the evolution of model checker input languages. Some model checkers, for example JPF and dSPIN, have already begun to incorporate BIR's non-determinism constructs for dynamic data since they dramatically increase modeling power without expanding the state-space.

Tool interoperability is a challenging but often underappreciated goal that potentially has significant benefits – especially for an emerging area such as software model-checking. Ideally, researchers should be able to leverage each other's tool-building efforts to avoid excessive duplication of effort. While BIR is by no means perfect, we hope that the effort reported here contributes to a dialogue among like-minded researchers regarding representations for software systems and specifications amenable to model-checking.

Acknowledgements

This work was supported primarily by NASA under grant NAG-02-1209 with additional support from DARPA/ITO's PCES program through AFRL Contract F33615-00-C-3044, from NSF under grants CCR-9703094, CCR-9708184, CCR-9896354 and CCR-9901605, by the U.S. Army Research Laboratory and the U.S. Army Research Office under agreement number DAAD190110564, and from the Formal Verification of Integrated Modular Avionics Software cooperative agreement, NCC-1-399, sponsored by Honeywell Technology Center and NASA Langley Research Center.

Robby, Corina Pasareanu, and Venkatesh Ranganath contributed to this work through numerous discussions about the design and formalization of BIR.

We would like to thank the anonymous reviewers of this paper for their helpful comments and suggestions.

Finally, the authors gratefully acknowledge the very significant contributions of Jay Corbett who is responsible for the initial design of BIR and the initial implementation of BIRC and the SPIN and SMV translators.

References

1. R. Alur, R. Grosu, and M. McDougall. Efficient reachability analysis of hierarchical reactive machines. *Proceedings 12th International Conference on Computer Aided Verification*, 2000.
2. E. Ashcroft and Z. Manna. Formalization of properties of parallel programs. *Machine Intelligence*, pages 17–41, 1972.
3. George S. Avrunin, James C. Corbett, , and Matthew B. Dwyer. Benchmarking finite-state verifiers. *International Journal on Software Tools for Technology Transfer*, 2(4):317–320, April 2000.
4. T. Ball and S. Rajamani. Bebop: a symbolic model-checker for boolean programs. In K. Havelund, John Penix, and Willem Visser, editors, *Proceedings of Seventh International SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*, pages 113–130. Springer-Verlag, 2000.
5. Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, Cesar Mu noz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In *Proceedings of the Fifth Langley Formal Methods Workshop*, June 2000.
6. A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV : a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.
7. A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. *Proceeding of International Conference on Computer-Aided Verification (CAV)*, 2002.

8. E.M. Clarke and E.A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. *Lecture Notes in Computer Science*, 131, 1981.
9. James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera : Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.
10. James C. Corbett, Matthew B. Dwyer, John Hatcliff, and Robby. Bandera : A source-level interface for model checking Java programs. In *Proceedings of the 22nd International Conference on Software E ngineering*, June 2000.
11. James C. Corbett, Matthew B. Dwyer, John Hatcliff, and Robby. Expressing checkable properties of dynamic systems: The bandera specification language. *International Journal on Software Tools for Technology Transfer*, 4(1), 2002.
12. Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. *Lecture Notes in Computer Science*, 952, 1995.
13. C. Demartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent Java programs. *Software - Practice and Experience*, 29(7):577–603, July 1999.
14. C. Demartini, R. Iosif, and R. Sisto. dSPIN : A dynamic extension of SPIN. In *Theoretical and Applied Aspects of SPIN Model Checking (LNCS 1680)*, September 1999.
15. Xianghua Deng, Matthew B. Dwyer, John Hatcliff, and Masaaki Mizuno. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *Proceedings of the 24nd International Conference on Software Engineering*, May 2002.
16. Matthew Dwyer, John Hatcliff, Roby Joehanes, Shawn Laubach, Corina Pasareanu, Robby, Willem Visser, and Hongjun Zheng. Tool-supported program abstraction for finite-state verification. *Proceedings of the 23rd International Conference on Software Engineering*, pages 177–187, 2001.
17. Matthew B. Dwyer, John Hatcliff, Roby Joehanes, Shawn Laubach, Corina S. Păsăreanu, Robby, Willem Visser, and Hongjun Zheng. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd International Conference on Software Engineering*, May 2001.
18. M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, May 1999.
19. Stefan Edelkamp, Alberto Lluch-Lafuente, and Stefan Leue. Directed model-checking with HSF-Spin. *Proceedings of the 8th International SPIN Workshop on Software Model Checking, Lecture Notes in Computer Science*, 2001.
20. R. Gerth, R. Kuiper, D. Peled, and W. Penczek. A partial order approach to branching time logic model checking. *Proceedings 3rd Israel Symposium on Theory on Computing and Systems*, pages 130–139, 1995.
21. Carlo Ghezzi and Mehdi Jazayeri. *Programming Language Concepts (3rd edition)*. John Wiley and Sons, 1997.
22. R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, 1996.
23. Patrice Godefroid. Partial-order methods for the verification of concurrent systems. *Lecture Notes in Computer Science*, 1032, 1996.
24. Patrice Godefroid. Model-checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 174–186, January 1997.

25. John Hatcliff, James C. Corbett, Matthew B. Dwyer, Stefan Sokolowski, and Hongjun Zheng. A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In *Proceedings of the 6th International Static Analysis Symposium (SAS'99)*, September 1999.
26. John Hatcliff, Matthew B. Dwyer, Corina S. Păsăreanu, and Robby. Foundations of the bandera abstraction tools. In *The Essence of Computation*, LNCS 1490, 2000.
27. John Hatcliff, Matthew B. Dwyer, and Hongjun Zheng. Slicing software for model construction. *Higher-order and Symbolic Computation*, 13(4), 2000.
28. K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 1999.
29. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Proceedings of the ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages*, Portland, Oregon, January 2002.
30. Gerard Holzmann. The spin model checker. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.
31. Gerard Holzmann and Doron Peled. An improvement in formal verification. *Formal Description Techniques*, pages 197–211, 1994.
32. Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.
33. Gerard J. Holzmann and Margaret H. Smith. Software model checking : Extracting verification models from source code. In *Proceedings of FORTE/PSTV'99*, November 1999.
34. Radu Iosif. Exploiting heap symmetries in explicit-state model checking of software. *Proc 16th IEEE International Conference on Automated Software Engineering*, pages 254–261, 2001.
35. Radu Iosif and Riccardo Sisto. dspin: A dynamic extension of spin. *Proc. 6th SPIN Workshop, Lecture Notes in Computer Science*, 1680:261–276, 1999.
36. Radu Iosif and Riccardo Sisto. Using garbage collection in model checking. *Proc. 7th SPIN Workshop, Lecture Notes in Computer Science*, 1885:20–33, 2000.
37. Roby Joehanes. *Incorporating UML State Charts into Bandera*. PhD thesis, Kansas State University, 2002. (Master of Science thesis).
38. T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Proceedings of the 7th International Static Analysis Symposium (SAS'00)*, 2000.
39. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Sun Microsystems, Inc., 1997.
40. Alberto Lluch-Lafuente, Stefan Edelkamp, and Stefan Leue. Partial order reduction in directed model checking. *Proceedings of the 9th International SPIN Workshop on Software Model Checking, Lecture Notes in Computer Science*, 2318, 2002.
41. L. Mounier M. Bozga, S. Graf. IF2.0: A validation environment for component-based real-time systems. In *Proceedings of the Fourteenth International Conference on Computer-Aided Verification (CAV 2002) (LNCS 2404)*, July 2002.
42. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1991.
43. S.S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

44. Rocco De Nicola and Frits Vaandrager. Three logics for branching bisimulation. *Proceedings of 5th Annual IEEE Symposium on Logic in Computer Science*, pages 118–129, 1990.
45. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *Proceedings of the 1th International Conference on Automated Deduction (LNCS 607)*, 1992.
46. Corina S. Pasareanu, Matthew B. Dwyer, and Willem Visser. Finding feasible counter-examples when model checking java programs. *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, Lecture Notes in Computer Science*, 2031, 2001.
47. Amir Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. *Current Trends in Concurrency, Lecture Notes in Computer Science*, pages 510–584, 1986.
48. Corina S. Păsăreanu, Matthew B. Dwyer, and Willem Visser. Finding feasible counter-examples when model checking abstracted Java programs. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*. Springer-Verlag, April 2001.
49. S. Stoller. Model-checking multi-threaded distributed Java programs. In K. Havelund, John Penix, and Willem Visser, editors, *Proceedings of Seventh International SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*, pages 224–244. Springer-Verlag, 2000.
50. Oksana Tkachuk. *Adapting Side-effects Analysis for Java Environment Generation*. PhD thesis, Kansas State University, 2003. (Master of Science thesis).
51. Oksana Tkachuk and Matthew B. Dwyer. Adapting side effects analysis for modular program model checking. In *Proceedings of the Fourth joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, September 2003.
52. Oksana Tkachuk, Matthew B. Dwyer, and Corina S. Păsăreanu. Automated environment generation for software model checking. In *Proceedings of the Fourth joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, October 2003.
53. Raja Valle-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java optimization framework. In *Proceedings of CASCON'99*, November 1999.
54. W. Visser, G. Brat, K. Havelund, and S. Park. Model checking programs. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, September 2000.
55. Eran Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *Proceedings of the ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages*, pages 27–40, jan 2001.

Appendix

A. Grammar

This section presents the syntax of the BIR language.

```
⟨system⟩          ::= 'system' ID '(' ')' { ⟨definition⟩ }
                      { ⟨thread⟩ } [ ⟨predicates⟩ ] 'end' ID ';'

⟨definition⟩      ::= ⟨constantdef⟩ | ⟨typedef⟩ | ⟨subtypedef⟩ | ⟨collectdef⟩
                      | ⟨globaldef⟩

⟨constantdef⟩     ::= 'const' CONSTANTID INT ';' | 'const' CONSTANTID
                      ⟨boolean⟩ ';'

⟨subtypedef⟩      ::= TYPEID 'extends' TYPEID ';'

⟨collectdef⟩      ::= COLLECTID ':' 'collection' [ '[' ⟨constant⟩
                      ']' ] 'of' TYPEID ';'

⟨globaldef⟩       ::= ID ':' ⟨type⟩ [ ':=' ⟨value⟩ ] ';'

⟨constant⟩        ::= CONSTANTID | INT

⟨type⟩            ::= ⟨typespec⟩ | TYPEID

⟨typedef⟩         ::= TYPEID '=' ⟨namedtypespec⟩ | TYPEID '=' ⟨typespec⟩

⟨typespec⟩        ::= 'boolean'
                   |  'range' ⟨constant⟩ '..' ⟨constant⟩
                   |  'lock' [ 'wait' ] [ 'reentrant' ]
                   |  'ref' TYPEID '{' ID { ',' ID } '}'
                   |  'array' '[' ⟨constant⟩ ']' 'of' ⟨type⟩

⟨namedtypespec⟩   ::= 'enum' '{' ⟨enumconst⟩ { ',' ⟨enumconst⟩ } '}'
                   |  'record' '{' { ID ':' ⟨type⟩ ';' } '}'

⟨enumconst⟩       ::= ID | ID '=' INT

⟨expr⟩            ::= ⟨value⟩
                   |  ⟨locktest⟩
                   |  ⟨threadtest⟩
                   |  '(' ⟨expr⟩ ')'
                   |  ⟨unop⟩ ⟨expr⟩
                   |  ⟨expr⟩ ⟨binop⟩ ⟨expr⟩
```

```
                  |  ⟨expr⟩ ‘.’ ID
                  |  ⟨expr⟩ ‘[’ ⟨expr⟩ ‘]’
                  |  ⟨expr⟩ ‘.’ ‘length’
                  |  ⟨expr⟩ ‘instanceof’ ID

⟨lhs⟩            ::= ID
                  |  ⟨lhs⟩ ‘.’ ID
                  |  ⟨lhs⟩ ‘[’ ⟨expr⟩ ‘]’
                  |  ⟨lhs⟩ ‘.’ ‘length’

⟨value⟩          ::= INT
                  |  ID
                  |  ⟨boolean⟩
                  |  ‘null’

⟨boolean⟩        ::= ‘true’ | ‘false’

⟨unop⟩           ::= ‘+’ | ‘-’ | ‘!’

⟨binop⟩          ::= ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘%’ | ‘&&’ | ‘||’ | ‘==’ | ‘!=’ |
                     ‘<’ | ‘>’ | ‘<=’ | ‘>=’

⟨locktest⟩       ::= ⟨locktestop⟩ ‘(’ ⟨lhs⟩ ‘)’

⟨locktestop⟩     ::= ‘lockAvailable’ | ‘hasLock’ | ‘wasNotified’

⟨threadtest⟩     ::= ‘threadTerminated’ ‘(’ ID ‘)’

⟨thread⟩         ::= [ ‘main’ ] ‘thread’ ID ‘(’ { ⟨param⟩ } ‘)’
                     { ⟨local⟩ } ⟨location⟩ { ⟨location⟩ } ‘end’ ID
                     ‘;’

⟨param⟩          ::= ID ‘:’ ⟨type⟩ ‘;’

⟨local⟩          ::= ID ‘:’ ⟨type⟩ [ ‘:=’ ⟨value⟩ ] ‘;’

⟨location⟩       ::= ‘loc’ ID ‘:’ [ ⟨liveset⟩ ] { ⟨transformation⟩ }

⟨liveset⟩        ::= ‘live’ ‘{’ ‘}’
                  |  ‘live’ ‘{’ ID { ‘,’ ID } ‘}’

⟨transformation⟩ ::= ‘when’ ⟨expr⟩ ‘do’ [ ‘invisible’ ]
                     ‘{’ { ⟨action⟩ } ‘}’ ‘goto’ ID ‘;’
```

```
⟨action⟩             ::= ⟨assignaction⟩
                       | ⟨choiceaction⟩
                       | ⟨lockaction⟩
                       | ⟨threadaction⟩
                       | ⟨printaction⟩
                       | ⟨assertaction⟩

⟨assignaction⟩       ::= ⟨lhs⟩ ‘:=’ ⟨expr⟩
                       | ⟨lhs⟩ ‘:=’ ‘new’ COLLECTID ‘;’
                       | ⟨lhs⟩ ‘:=’ ‘new’ COLLECTID ‘[’ ⟨expr⟩ ‘]’ ‘;’

⟨choiceaction⟩       ::= ⟨lhs⟩ ‘:=’ ‘internChoose’ ‘(’ ⟨value⟩ { ‘,’ ⟨value⟩
                         } ‘)’ ‘;’
                       | ⟨lhs⟩ ‘:=’ ‘externChoose’ ‘(’ ⟨value⟩ { ‘,’ ⟨value⟩
                         } ‘)’ ‘;’
                       | ⟨lhs⟩ ‘:=’ ‘forall’ ‘(’ ID ‘)’ ‘;’
                       | ⟨lhs⟩ ‘:=’ ‘reachable’ ‘(’ ID ’,’ ⟨expr⟩ ‘)’ ‘;’

⟨lockaction⟩         ::= ⟨lockop⟩ ‘(’ ⟨lhs⟩ ‘)’ ‘;’

⟨lockop⟩             ::= ‘lock’ | ‘unlock’ | ‘wait’ | ‘unwait’ | ‘notify’
                         | ‘notifyAll’

⟨threadaction⟩       ::= [ ⟨lhs⟩ ‘:=’ ] ‘start’ ‘(’ ID [ ‘,’ ⟨args⟩ ] ‘)’ ‘;’
                       | ‘exit’ ‘;’

⟨args⟩               ::= ⟨expr⟩ { ‘,’ ⟨expr⟩ }

⟨printaction⟩        ::= ‘println’ ‘(’ [ ⟨printargs⟩ ] ‘)’ ‘;’

⟨printargs⟩          ::= ⟨printarg⟩
                       | ⟨printarg⟩ ‘,’ ⟨printargs⟩

⟨printarg⟩           ::= STRING | ID

⟨assertaction⟩       ::= ‘assert’ ‘(’ ⟨expr⟩ ‘)’ ‘;’

⟨predicates⟩         ::= ‘predicates’ { ⟨predicate⟩ }

⟨predicate⟩          ::= ID ‘=’ ⟨predexpr⟩ ‘;’

⟨predexpr⟩           ::= ⟨threadLocationTest⟩ | ⟨remoteReference⟩

⟨threadLocationTest⟩ ::= ID ‘[’ ⟨lhs⟩ ‘]’ ‘@’ ID
```

$\langle remoteReference \rangle$ ::= ID '[' $\langle lhs \rangle$ ']' ':' $\langle lhs \rangle$

B. Proofs

We need the following lemma for the rest of the proofs. The proof uses the fact that we cannot have an invisible transformation originating from and ending at the same location in a thread declaration.

LEMMA 1. *Let $s, s' \in \widehat{State}$ and $t, t' \in ThreadId$. If $s \overset{t}{\mapsto}_{inv} s'$ and $s \overset{t'}{\mapsto} s'$ then $t = t'$.*

Proof: Let $s = \langle G, H, T \rangle$, $s' = \langle G', H', T' \rangle$ and let $\langle$`when (e) do [invisible]` $\{$`a`$_1, \ldots,$ `a`$_n\}$ `goto m`$\rangle \in Code(l)$ be the transformation from the precondition of rule (6) that makes $s \overset{t'}{\mapsto} s'$ true. By rule (6) $T(t') = \langle l, n, active, \sigma \rangle$, $T'(t') = \langle m, n', s', \sigma' \rangle$ and $\forall t'' \neq t'$ $T(t'') = T'(t'')$. We assume $t \neq t'$ and prove a contradiction. We have from $t \neq t'$ that $T(t) = T'(t)$. Let $\langle$`when (e) do invisible` $\{$`a`$_1, ...,$ `a`$_n\}$
`goto` $\overline{\mathtt{m}}\rangle \in Code(\overline{l})$ be the transformation that triggers $s \overset{t}{\mapsto}_{inv} s'$. Then, according to rule (6), we have $T(t) = \langle \overline{l}, \overline{n}, active, \overline{\sigma} \rangle$, $T'(t) = \langle \overline{m}, \overline{n'}, \overline{s'}, \overline{\sigma'} \rangle$, and since $T(t) = T'(t)$ we have $\overline{l} = \overline{m}$. This contradicts the syntactic restriction that no invisible transformation originates and ends in the same control location. □

The following proof uses the syntactic restriction that it is illegal to have a visible and an invisible transformation originating from and ending at the same location.

PROPOSITION 1. *For any $s, s' \in State$ and $t_1, t_2 \in ThreadId$, it is not the case that both $s \overset{t_1}{\mapsto}_{inv} s'$ and $s \overset{t_2}{\mapsto}_{vis} s'$ hold.*

Proof: There are two cases:

i) If $s, s' \in \widehat{State}$, then let $s = \langle G, H, T \rangle$ and $s' = \langle G', H', T' \rangle$. We prove by contradiction, assuming that $s \overset{t_1}{\mapsto}_{inv} s'$, $s \overset{t_2}{\mapsto}_{vis} s'$ and letting $\langle$`when (e) do invisible`$\{$`a`$_1, ...,$ `a`$_n\}$ `goto m`$\rangle \in Code(l)$ be the transformation from the precondition of rule (6) that makes $s \overset{t_1}{\mapsto}_{inv} s'$ true. If $s \overset{t_2}{\mapsto}_{vis} s'$, by Lemma 1 we obtain $t_2 = t_1$. Now let $\langle$`when (e`$'$`) do` $\{$`a`$'_1, ...,$ `a`$'_n\}$ `goto m`$\rangle \in Code(l)$ be the transformation that makes $s \overset{t_2}{\mapsto}_{vis} s'$ true, according to rule (6). Clearly, the existence of both transformations between control locations l and m is a violation of the syntactic restriction regarding the presence of both visible and invisible transformations between two control locations.

ii) If $s \in \widehat{State}$ and $s' \in \{ErrorState, LimitState\}$ it cannot be the case that $s \overset{t_1}{\mapsto}_{inv} s'$, since $\mapsto_{inv}$ is defined as the least relation satisfying rule (7).

□

The soundness proof for transition labeling will be carried out using the rules in Figure 16.

PROPOSITION 2. *For any $s, s' \in State$ and $u, u', v, v' \in ThreadId_\epsilon$, it is not the case that both $\langle s, u\rangle \rightsquigarrow_{vis} \langle s', u'\rangle$ and $\langle s, v\rangle \rightsquigarrow_{inv} \langle s', v'\rangle$ hold.*

Proof: As $\hookrightarrow$ is the least relation meeting the rules (10, 11, 12), $\langle s, u\rangle \hookrightarrow_{vis} \langle s', u'\rangle$ holds because either i) $s \overset{t}{\mapsto}_{vis} s'$ or ii) $s \overset{t}{\mapsto}_{inv} s'$ and $out(s', t) = \emptyset$. The (i) case is ruled out by the fact that $\langle s, v\rangle \hookrightarrow_{inv} \langle s', v'\rangle$ which can only be true due to $s \overset{t'}{\mapsto}_{inv} s'$ for some $t' \in ThreadId_\epsilon$. According to Proposition 1, having both $s \overset{t}{\mapsto}_{vis} s'$ and $s \overset{t'}{\mapsto}_{inv} s'$ is a contradiction. For the (ii) case, as we can only have $\langle s, v\rangle \hookrightarrow_{inv} \langle s', v'\rangle$ due to an application of rule (10), it is the case that $out(s', t) \neq \emptyset$ and $s \overset{t'}{\mapsto}_{inv} s'$ for some $t' \in ThreadId_\epsilon$. Then by Lemma 1 we have $t = t'$. But this is clearly in contradiction with the fact that $out(s', t) = \emptyset$. □

The following proposition shows that the representative function $h : S_R \to S$ is indeed well defined.

PROPOSITION 3. *Let $[s]_R \subseteq S$ be an equivalence class w.r.t. R. If $s_1, s_2 \in [s]_R$ such that for no $s'_1 \in [s]_R$ we have $s'_1 \Rightarrow s_1$ and for no $s'_2 \in [s]_R$ we have $s'_2 \Rightarrow s_2$, then $s_1 = s_2$.*

Proof: By contradiction, assume that $s_1 \neq s_2$. Since $s_1, s_2 \in [s]_R$ and $s_1 \neq s_2$ then either $s_1 \Rightarrow s_2$ or $s_2 \Rightarrow s_1$, by the definition of R. But either case contradicts the hypothesis. □

The following proposition shows that M and M_h are branching bisimilar. This is done by showing first that R is a branching bisimulation. Since R is total on both S and $h(S_R)$, the result follows immediately.

PROPOSITION 4. *For any $s, s', t \in S$, if sRs' and $s \longrightarrow t$ then either:*

a) $s \longrightarrow_{inv} t$ and tRs', or

b) there exist $s_1, t' \in S$ such that $s' \Rightarrow s_1 \longrightarrow t'$ and sRs_1 and tRt'.

Proof: By definition, sRs' is because either (1) $s = s'$, (2) $s \Rightarrow s'$ or (3) $s' \Rightarrow s$. The first case meets trivially condition (b). Assume now that $s \Rightarrow s'$. Then, for some $s'' \in S$ we have $Out(s) = \{s''\}$, $s \longrightarrow_{inv} s''$ and $s'' \Rightarrow s'$. The only possibility is to have $t = s''$ and therefore $s \longrightarrow_{inv} t$ and $t \Rightarrow s'$. This leads to tRs' which satisfies condition (a). In the third case we have $s' \Rightarrow s$ and since $s \longrightarrow t$, condition (b) is immediately satisfied. □

C. NuSMV Translation

NuSMV [6] is a symbolic model checker that uses BDDs to encode the set of reached states, and the transition relation is represented as a predicate transformer. In this section we briefly sketch the translation of BIR to NuSMV.

A first difference with respect to the SPIN translation is that dynamic threads are not considered. Instead, we assume a set *ThreadName* of thread names that are either idle or active. In the beginning, all threads are idle except for a designated main thread. An idle thread can be activated, but there are no means of generating fresh names for threads. To some extent, this limitation can be overcome by over-approximating the maximum number of threads that the program will create and declaring enough names. Note that such an over-approximation is not always possible.

Variables: For each thread name $T \in ThreadName$ we declare two global variables, T_loc recording its current location, and T_active indicating whether the thread is active. Local variables of each thread are translated in NuSMV by prefixing their names with the name of their enclosing thread.

Among the global variables, BIR collections request special attention. A collection X having size k is translated into $2k$ distinct variables i.e., for each $i \in \{0, \ldots, k-1\}$: X_inuse$i$ of type boolean, indicating whether the i-th collection slot is in use, and, X_inst$i$ represents a particular instance, according to its type. A similar scheme is used for the translation of array types. Records are flattened by prefixing each field with the name of the record type.

A BIR variable X having a reference type is translated into a pair of variables: X_refIndex identifies the collection pointed to, and X_instNum indicates the index of the specific instance inside the collection.

The order of variables plays an important role in NuSMV, since the sizes of the BDDs used to represent the set of reachable states greatly depend on this ordering. Even though there is no efficient way to determine an optimal variable order, heuristics proposed in the literature [1] suggest using the hierarchy in the system to order variables. Our translation defines a partial order in which all global variables, record fields, array elements and thread local variables are at the top of the order.

Transitions: The global transition relation is given by a boolean formula of the form:

$$\mathit{TRANS} = \bigwedge_{T \in \mathit{ThreadNames}} \mathit{Trans}(T) \wedge \bigwedge_{v \in \mathit{VarNames}} \mathit{Trans}(v)$$

where $\mathit{Trans}(T)$ defines the local behavior of a thread and $\mathit{Trans}(v)$ defines the behavior of a global variable. Thus to define the transition relation, it is sufficient to define the transitions of each thread and each variable.

The interleaving semantics of a multithreaded program is captured in the synchronous execution mode of NuSMV by introducing a designated variable *running*, of a special type `thread_id`, whose value is unconstrained, and therefore updated non-deterministically. Only *running* may take a transition, while all other threads idle. This is captured in the following relation:

$$\begin{aligned}\mathit{Trans}(T) \;&=\; \left(T\_\mathrm{loc} = T'\_\mathrm{loc} \wedge \bigwedge_{v \in \mathit{Loc(T)}} v' = v\right) \vee \\ &\quad \bigvee_{t \in \mathit{Tr(T)}} \left(\mathit{taken(t)} \wedge \bigwedge_{v \in \mathit{Loc(T)}} \mathit{update}(v,t)\right)\end{aligned}$$

where *taken(t)* is a shorthand for:

$$\mathit{running} = T \wedge \mathit{guard(t)} \wedge T\_\mathrm{loc} = \mathit{source(t)} \wedge T\_\mathrm{loc}' = \mathit{target(t)}$$

and

$$\mathit{update}(v,t) = \begin{cases} v' = e & \text{if } t \text{ assigns } e \text{ to } v \\ v' = v & \text{if } t \text{ does not assign } v\text{, but } v \text{ is live at } \mathit{target}(t) \\ 1 & \text{otherwise} \end{cases}$$

For a thread T, *Tr(T)* denotes the set of transformations, whereas for a transformation t, *source(t)* denotes its source location and *target(t)* stands for its target location.

Note that dead variables are left unconstrained by the update formula. In practice this has shown important reductions in the size of the transition relation BDD.

Expressions: Most BIR arithmetic operators have an NuSMV counterpart, but dereferencing requires special treatment. We have used the NuSMV *case* selection which, for the purposes of this presentation is abbreviated as: $\bigsqcup_{i=1}^{n}(x : y) =$ case $x_1 : y_1;\ x_2 : y_2;\ \ldots;\ x_n : y_n;$ esac. The result of the expression is the y_i value for the first x_i expression that evaluates to true, or 0 if all x_i are false.

When a reference variable R is dereferenced, we generate nested case expressions to select the correct collection and instance.

$$\bigsqcup_{r \in \mathit{Targets}(R)} \left(R\_\mathrm{refIndex} = r : \bigsqcup_{i=0}^{\mathit{Size}(r)-1} (R\_\mathrm{instNum} = i : \mathit{Name}(r)\_\mathrm{inst}i) \right)$$

where *Targets*(R) is the set of target collections to which R could refer (determined by its declared type), *Size*(r) is the size of the collection r, and *Name*(r) is the name of collection r. If the target r is a singleton, we can omit the inner case expression i.e., there is no instance to select.