Bogor: A Flexible Framework for Creating Software Model Checkers

Robby
Kansas State University
robby@cis.ksu.edu

Matthew B. Dwyer
University of Nebraska, Lincoln
dwyer@cse.unl.edu

John Hatcliff
Kansas State University
hatcliff@cis.ksu.edu

Abstract

Model checking has proven to be an effective technology for verification and debugging in hardware and more recently in software domains. With the proliferation of multi-core architectures and a greater emphasis on distributed computing, model checking is an increasingly important software quality assurance technique that can complement existing testing and inspection methods.

We believe that recent trends in both the requirements for software systems and the processes by which systems are developed suggests that domain-specific model checking engines *may be more effective than general purpose model checking tools. To overcome limitations of existing tools which tend to be monolithic and non-extensible, we have developed an extensible and customizable model checking framework called Bogor. In this article, we summarize how Bogor provides direct support for modeling object-oriented designs and implementations, how its modeling language and algorithms can be extended and customized to create domain-specific model checking engines, and how Bogor can be deployed in broader software development context contexts in conjunction with complementary quality assurance techniques.*

1. Motivation

Temporal logic model checking [11] is a powerful framework for reasoning about the behavior of computing systems. Model checkers exhaustively check a finite-state model of a system for violations of a system requirement formally specified as a formula in some temporal logic (e.g., Linear Temporal Logic (LTL) [37]), as an automata, or as a collection of assertions. The system model may be constructed by hand or derived automatically from other artifacts such as source code, executables, or higher-level designs using various abstraction techniques. Model checking has been so successful in the hardware domain that virtually all major chip manufacturers employ it as a central part of the quality assurance process.

In recent years, the use of model checking has been growing in the software domain and it has been been applied, in various forms, to reason about a wide-variety of software artifacts. For example, model checking frameworks have been applied to reason about software process models (e.g., [33]), different families of software requirements models (e.g., [8]), architectural frameworks (e.g., [24]), design models, (e.g., [26]), and system implementations (e.g., [4, 7, 12, 25]).

Several additional factors are driving interest in using model checking to complement existing software quality assurance techniques. The widespread use of concurrency primitives in programming languages like Java and C#, the proliferation of multicore processors, and the growing dependence on large-scale distributed computing applications means that the vast majority of programmers will be programming in concurrent/multi-threaded settings. The potential of model checking technology for: (a) detecting coding flaws that are hard to detect using existing quality assurance methods (e.g., such as bugs that arise from unanticipated interleavings in concurrent and distributed programs), and (b) verifying that system models and implementations satisfy crucial temporal properties and other light-weight specifications has led a number of international corporations and government research labs such as Microsoft [4], IBM [5], Lucent [25], NEC [32], NASA [7], and Jet Propulsion Laboratories (JPL) [29] to fund their own software model checking projects.

1.1. Trends that motivate domain-specific model checking

The effectiveness of the software model checking applications cited above has in most cases relied on the fact that the researchers applying model checking had a detailed knowledge of the model checking tool being applied and understood very clearly (perhaps as the result of extensive experimentation) how to (a) map the problem being modeled to the tool in a manner that would draw on the strengths of the tool, and (b) how to configure the tool for optimal performance. In some cases, researchers were not satisfied

with the results of mapping their problem to an available configuration of an existing tool, and they found it necessary to study an *existing* model checking framework in detail in order to customize it [8, 13]. In other cases, researchers had the insight that they needed to develop a *new* framework targeted to the semantics of a family of artifacts (e.g., Java programs [7]) or a specific domain (e.g., device drivers [4]).

We believe there are a number of trends both in the requirements of computing systems and in the processes by which they are developed that will drive persons and organizations interested in applying model checking technology to rely increasingly on customization/adaptation of existing tool frameworks or construction of new model checking tools.

- **Model-driven development:** Increasing emphasis on using a rich collection of interlinked models at a variety of levels of abstraction for not only software design but also software implementation will drive researchers to look beyond model checkers with fixed input languages that target a single level of abstraction. Current technology allows model checking to be applied at the source code and byte code (or machine code) level as well as higher level models – but for the most part, different tools with different characteristics are used for these different levels of abstraction. We envision a single integrated framework in which artifacts at all levels of abstraction are checkable, in which relationships such as refinement/conformance are established by model checking collections of artifacts at different levels of abstraction, and in which verification models are formed by composing: (a) analyzable models from which code is automatically generated, (b) manually written source that was not amenable to auto-generation, and (c) high-level specifications of cross-cutting *aspects* such as synchronization or event delivery policies.

- **Product-line architectures and reusable middleware infrastructure:** Large-scale distributed systems are increasingly built on top of middleware frameworks such as CORBA and .NET and use software product-line architectures that encourage the reuse of software components and infrastructure across applications. The size (often 100's of thousands of lines of code) and the complexity of this infrastructure (which often contains extensive use of object-oriented design patterns and complex heap structures) will make it difficult to apply existing techniques for model-extraction from source code, but the high degree of reuse of this infrastructure will make it feasible to construct custom-built verification model components and checking engines that are tailored to those frameworks. Moreover, as greater automation is sought in development and validation processes, developers will have a greater difficulty applying general purpose model checking tools which require a higher degree of configuration and interaction, and product-line architects will increasingly seek to reduce developer involvement by building model checking infrastructure that is dedicated to a particular software architecture, domain, and development process.

- **Synergistic blending of automated quality assurance techniques:** Each quality assurance technique has certain strengths and weaknesses. Static analyses achieve full coverage and scale well but are limited to a particular set of properties and generates false alarms due to approximations inherent in each analysis. Testing is flexible and well-understood, but creating test cases is burdensome and achieving desired coverage levels (especially for concurrent programs) is often difficult. Run-time monitoring and dynamic analysis are highly precise methods, but introduce overhead and only cover execution paths associated with particular program runs. Model-checking techniques provide coverage of concurrent behaviors and scheduler non-determinism but suffer from scalability problems. Symbolic execution can provide property checking without test cases and generate path conditions and test cases, but also suffers from scalability issues and cannot deal well with concurrency.

  There is expanding interest in exploring how these techniques can be synergistically integrated. For example, dynamic analysis techniques can discover likely invariant properties concerning aliasing, locking, and object-ownership that would be burdensome to specify manually, and soundness of these properties (realized as program-level annotations) can be established with complete coverage using static analysis [1]. Model checking engines can be used on very abstract program models to generate test plans for model-driven testing [38]. Statically-computed object "escape" and points-to information can be used to drive partial-order optimizations in model checking [18]. These integrations benefit from the ability to "open up" tool internals and thereby enable direct interaction with the primary model checking data structures and algorithms.

- **Sheer variety of application domains and computation models:** As software is embedded in an ever broadening range of devices and as software increases in scale, verification experts will increasingly move away from general purpose solutions as they seek to achieve scalability by developing search algorithms, state-storage approaches, and model reduction strate-

gies that leverage properties of a particular domain. For example, we have observed that reduction methods designed for a particular application domain (e.g., Java programs) may be ineffective for a different application domain such as avionics systems where timing issues play a greater role. Conversely, we have found that the specialized quasi-cyclic structure of computation in a certain class of avionics systems enables dramatic optimizations in state space search and state storage strategies that are customized for this particular domain [14, 19].

Thus, there is a need for model checking tools that support customization and extensibility that are easily embedded or encapsulated in larger development tools, and that can be flexibly arranged in the workflow of realistic software development processes. Existing model checkers, including widely used tools such as SPIN [29], FDR2 [22], and NuSMV [10], are designed to support a fixed input language using a fixed collection of state-space representation, reduction and exploration algorithms. The capabilities of these tools has evolved over time, but that evolution has been limited to the capabilities that the tool developer themselves found useful or desirable. Moreover, most existing tools do not provide direct support for features found in object-oriented software systems. Recent versions of the SPIN model checker allow one to include C source directly in the Promela modeling language. Even though this does allow for a degree of extensibility, one might hope for more comprehensive extensibility mechanisms that are part of the overall design of the model checking framework.

While it is possible to continue the practice of cannibalizing and modifying existing model checkers, or building new model checkers from scratch, the level of knowledge and effort required for such activities currently prevents many *domain* experts who are not necessarily experts in model checking from successfully applying model checking to software analysis. Even though experts in different areas of software engineering have significant domain knowledge about the semantic primitives and properties of families of artifacts that could be brought to bear to produce cost-effective semantic reasoning via model checking, in order to make effective use of model checking technology these experts should not be required to build their own model checker or to pour over the details of an existing model checker implementation while carrying out substantial modifications.

2. Bogor: a customizable and extensible framework

To meet the challenges of using model checking in the context of current trends in software development outlined above, we have constructed an extensible and highly modular explicit-state model checking framework called Bogor [39, 43]. Using Bogor, we seek to enable more effective incorporation of domain knowledge into verification models and associated model checking algorithms and optimizations, by focusing on the following principles.

- **Direct Support of Object-Oriented Languages:** Bogor provides a rich base modeling language including features that allow for dynamic creation of objects and threads, garbage collection, virtual method calls and exception handling. For these primitives, we have extended Bogor's default algorithms to support state-of-the-art model reduction/optimization techniques that we have developed [18, 40] for object-oriented software that use existing techniques such as collapse compression [29], heap symmetry [30], thread symmetry [6], and partial-order reductions.

- **Extensible Modeling Language:** Bogor's modeling language can be extended with new primitive types, expressions, and commands associated with a particular domain (e.g, multi-agent systems, avionics, and security protocols) and a particular level of abstraction (e.g., design models, source code, and byte code)

- **Open Modular Architecture:** Bogor's well-organized module facility allows new algorithms (e.g., for state-space exploration, and state storage) and new optimizations (e.g., heuristic search strategies, and domain-specific scheduling) to be easily swapped in to replace the default model checking algorithms.

- **Design for Encapsulation:** Bogor is written in Java and comes wrapped as a plug-in for *Eclipse* – an open source and extensible universal tool platform from IBM. This allows Bogor to be deployed as a standalone tool with a rich graphical user interface and a variety of visualization facilities, or encapsulated within other development or verification tools for a specific domain.

- **Pedagogical Materials:** Even with a tool that is designed for extensibility, creating customizations requires a significant amount of knowledge about the internal architecture. To communicate this knowledge, we have developed an extensive collection of tutorial materials and examples. Moreover, we believe that Bogor is an excellent pedagogical vehicle for teaching foundations and applications of model checking because it allows students to see clean implementations of basic model checking algorithms and to easily enhance and extend these algorithms in course projects. Accordingly, we have developed a comprehensive collection of course materials [44] that have already been

used in graduate level courses on model checking at several institutions.

In short, Bogor aims to be not only a robust and feature-rich software model checking tool that handles the language constructs found in modern large-scale software system designs and implementations, it also aims to be a model checking *framework* that enables researchers and engineers to create families of domain-specific model checking engines.

3. Bogor's support for object-oriented language features

Bogor checks systems specified in a revised version of the Bandera Intermediate Representation (BIR) [31]. The previous version of BIR was designed to be an intermediate language used by our Bandera tool set [12] for translating Java programs to the input languages of existing model checkers such as SPIN. Thus, this earlier version provided direct support for modeling Java features such as threads, Java locks (supporting wait/notify), and a bounded form of heap allocation. To facilitate the construction of translators to back-end model checkers like SPIN, BIR control-flow and actions are stated in a *guarded command* format which is quite close to the format used to specify systems in model checker input languages like Promela.

Our experience with Bandera and other tools such as JPF [7] and dSpin [13] has led us to conclude that software model checking can be more effectively supported by a new infrastructure that has at its core an extensible model checker that is designed to support software directly rather than relying on translations to model checkers that do not provide direct support for modeling many of the language features found in modern software. As part of this transition to a new infrastructure, we have revised the definition of BIR to include a number of new features such as the BIR extension mechanism, generic types and polymorphic functions, type-safe function pointers, virtual function/method tables, and exceptions.

BIR comes in two flavors: a higher-level language with structured control-flow statements often used in hand-coded models, and a lower-level language with explicit control locations and explicit control successors (e.g., specified with goto statements) often used as the target of automatic model compilers.

Figure 1 provides a simple list-manipulating system that illustrates features of high-level BIR.

- Lines 3 and 8 illustrate the use of BIR records to represent the data portions of objects. Object (line 3) represents the top element in an object inheritance hierarchy and includes no fields. ListNode (lines 8 extends Object to add fields used to form a linked list.

```
1  system LanguageFeatures
2  {
3    record Object {}
4
5    throwable record NullPointerException {}
6    throwable record IllegalArgumentException {}
7
8    record ListNode extends Object {
9      ListNode next;
10     int      data;
11   }
12
13   function createNode(int data, ListNode tail)
14            returns ListNode {
15     ListNode n;
16
17     n := new ListNode;
18     n.data := data;
19     n.next := tail;
20     return n;
21   }
22
23   function sumAllElementsOver5(ListNode head)
24            returns int {
25     int firstElementData;
26     int total;
27
28     // provoke NPE if bad parameter passed in
29     firstElementData := head.data;
30
31     // iterative loop...
32     while head != null do
33       head := head.next;
34
35       // guarded choice
36       choose
37         when <head.data > 5> do
38           total := total + head.data;
39         else do skip;
40       end
41     end
42
43     return total;
44   }
45
46   active thread MAIN() {
47     NullPointerException npe;
48     IllegalArgumentException iae;
49
50     ListNode n;
51     int length;
52
53     try
54       // first a call that succeeds
55       length := sumAllElementsOver5(
56                   createNode(
57                       5,
58                       createNode(
59                           7,
60                           null)));
61
62       // now provoke an exception that's handled below
63       length := sumAllElementsOver5(null);
64       catch (IllegalArgumentException iae)
65         skip;
66       catch (NullPointerException npe)
67         skip;
68     end
69   }
70 }
```

Figure 1. Illustration of High-level BIR Language Features

```
1  function sumAllElementsOver5(ListNode head)
2          returns int {
3
4    int firstElementData;
5    int total;
6    boolean temp$0;
7    int temp$1;
8
9    loc loc0:
10     do {
11       firstElementData := head.data;
12     } goto loc1;
13
14   loc loc1:
15     do {
16       temp$0 := head != null;
17     } goto loc2;
18
19   loc loc2:
20     when temp$0 do {
21     } goto loc3;
22     when !(temp$0) do {
23     } goto loc10;
24
25   loc loc3:
26     do {
27       head := head.next;
28     } goto loc4;
29
30   loc loc4:
31     when head.data > 5 do invisible {
32     } goto loc5;
33     when !((head.data > 5)) do invisible {
34     } goto loc8;
35
36   loc loc5:
37     do {
38       temp$1 := head.data;
39     } goto loc6;
40
41   loc loc6:
42     do {
43       total := total + temp$1;
44     } return total;
45   }
```

Figure 2. Illustration of Low-level BIR Language Features

- Lines 5 and 6 illustrate the use of BIR records to implement the data portion of an exception (in these cases, the exceptions contain no data fields).
- Lines 13–21 and Lines 23–44 define functions createNode and sumAllElementsOver5. createNode illustrates the dynamic creation of a list node. sumAllElementsOver5 uses several high-level control constructs.
- The system main thread at lines 46–69 illustrates the use of BIR's exception mechanism.

Figure 2 presents low-level BIR corresponding to the definition of sumAllElementsOver5 in Figure 1. High-level and low-level BIR can be freely mixed within a program model.

In contrast to the input languages used in almost all other model checkers (including SPIN [29], NuSMV [10], SLAM [3], and BLAST [28]), BIR supports unbounded dynamic creation of both thread and heap objects with automatic reclamation by garbage collection. Moreover, BIR provides a state-of-the-art canonical heap representation that seems essential for effective checking of highly dynamic concurrent software systems. Such systems generate many heap instances that differ in the relative position of allocated objects but that are actually *observationally equivalent* (i.e., the heap instances can not be distinguished by any Java memory operations). Due to positional differences of object placement in heaps, conventional representations of heaps as, for example, arrays of memory cells would yield different states for these heaps. However, Bogor's canonical heap representation (based on work by Iosif on dSpin [13]) ensures that heaps that are observationally equivalent map to the same state. This dramatically reduces the number of states generated when checking highly dynamic systems.

```
1  extension Set for myPackage.SetModule {
2      // declare a new type identifier (Set.type)
3      typedef type<'a>;
4
5      // create a set with zero or more elements
6      expdef Set.type<'a> create<'a>('a ...);
7
8      // non-deterministically choose an element
9      expdef 'a chooseElement<'a>(Set.type<'a>);
10
11     // check for emptiness
12     expdef boolean isEmpty<'a>(Set.type<'a>);
13
14     // check predicate over all elements
15     expdef boolean forAll<'a>('a -> boolean, Set.type<'a>);
16
17     // add an element to a set
18     actiondef add<'a>(Set.type<'a>, 'a);
19
20     // remove an element from a set
21     actiondef remove<'a>(Set.type<'a>, 'a);
22 }
```

Figure 3. Bogor Set Extension Declaration (excerpts)

4. Bogor's extensible modeling language

Bogor's extension facility allows modelers to add new types, expressions, and commands to the base modeling language. To create an extension, one first writes an extension declaration that specifies the new symbols and associated arities to be introduced into the name-spaces for types, expressions, and actions. Next, a Java package is written to implement the semantics of each of the new types, expressions, and actions specified in the extension declaration.

For example, Figure 3 illustrates an extension declaration for a *set* abstract data type. Line 1 gives the name of the extension (Set) and the name of the Java package that con-

```
1  // declare some record types representing two kinds
2  // of resources
3  record Resource { boolean isFree; }
4  record Disk extends Resource { }
5  record Display extends Resource { }
6
7  // declare a resource pool represented as a set
8  Set.type<Resource> resourcePool;
9
10 // declare a resource variable
11 Resource resource;
12 ...
13 loc loc1:
14   do { // create the pool and creates two processes
15      resourcePool := Set.create<Resource>
16      (new Disk, new Disk, new Display);
17      } goto loc1;
18
19 loc loc2: live {resource}
20   when !Set.isEmpty<Resource>(resourcePool)
21   do { // choose an element and remove it
22        resource := Set.choose<Resource>
23              (resourcePool);
24        Set.remove<Resource>(resourcePool,
25                             resource);
26     } goto loc3;
27
28 loc loc3: live {resource}
29     do { // add the resource back to pool
30          Set.add<Resource>(resourcePool,
31                            resource);
32        } return;
```

Figure 4. Bogor Set Extension Use (excerpts)

tains its implementation (myPackage.SetModuleSet). Line 3 declares a polymorphic type to represent the type of set elements. Lines 6, 9, and 12 define new expressions to create a set with the supplied arguments as elements (here "..." specifies that the expression can take a variable number of arguments), to non-deterministically pick an element from the set (chooseElement), and to test a set for emptiness (isEmpty). The forAll expression in line 15 illustrates a higher-order operation that returns *true* if a supplied predicate (first argument) holds for each element of a given set (second argument). Lines 18 and 21 define actions that add and remove elements from a set. Figure 4 provides some fragments of a BIR model that illustrate the use of the set extension.

It is important to note that an extension declaration does not extend the BIR grammar, but only adds names to the set of names of built-in expressions, actions, etc. This means that the developer does not need to extend the parser or other syntactic support facilities. Rather, in the Java package implementing the set, the developer traverses the extension structure and implements the extension semantics using well-defined APIs for BIR's existing abstract syntax trees and Bogor model checker components. The Bogor web site [43] provides an extended tutorial on how to implement the set extension described above.

Extensions provide a convenient way to realize domain-specific abstractions of software components or layers, and to address the input language gap for modeling domain-specific software artifacts. Often times, there are component/layers of the software that have a significant amount of state that is *irrelevant* to the properties being checked. Rather than maintaining a complete implementation of a software component/layer using BIR's variables, the Java package implementing the extension can hold the state associated with the complex component/layer and only expose as much as is relevant at the BIR level. Since only the BIR variables are held in Bogor's state vector during model checking, this can dramatically reduce the costs of representing portions of a software system.

Hiding complex portions of the state in this manner is not novel. For example, when modeling communication protocols using SPIN [29], channel data types (`chan`) are often used to model message-passing channel in networks. This can be done because the specific implementation of the network channels is *irrelevant* with respect to the properties being checked. The properties are usually only concerned with the functional behavior of the channels, i.e., rendezvous, asynchronous, synchronous, and sending and receiving messages. Furthermore, properties are usually only concerned with a channel's *abstract* states, i.e., the specific channel implementation state, for example, the state of message retransmission protocols used to provide the channel service, does not need to be exposed in the model state. What is novel about BIR is that it does not a priori hard-code such abstraction mechanisms for a particular domain – rather, BIR's extension facility provides an open-ended mechanism for adding any number of domain-specific abstractions.

We now overview two substantive extensions that highlight the power of this facility.

Modeling real-time CORBA Component Designs In our work on the Cadena development environment (also built in Eclipse) [26] for designing large-scale distributed real-time embedded systems built using the CORBA Component Model (CCM), we have extended Bogor's modeling language to include APIs associated with CCM components and an underlying real-time CORBA event service. [14, 19]. This enables BIR models to easily create and manipulate abstractions of CCM components. Figure 5 shows a very small excerpt of the BIR extension that enables creation and manipulation of components.

Using BIR CORBA event channel API extensions in component behavior models enables a close correspondence to the structure of actual CCM component business logic implementations. Without such extension facilities, the semantics of the CORBA event channel would need to be coded directly in BIR – which would result in a huge blowup in both the size and complexity of the BIR code

```
1  extension CAD for CADModule {
2    typedef Event;
3    typedef Component;
4    typedef Port;
5
6    // constructor operator
7    expdef CAD.Component createComponent(string name);
8
9    // hook an object onto a component as
10   // a named port
11   actiondef registerPort(CAD.Component c,
12                          CAD.Port p,
13                          string name);
14
15   // get all the subscribers to an event-
16   // producing port
17   expdef 'a[] getSubscribers<'a>(
18                       CAD.Component c,
19                       String eventPort);
20 }
```

Figure 5. Bogor Cadena Assembly Description Extension (excerpts)

```
1  // define an integer tag representing each Java byte code
2  const Op { ACONST_NULL = 1;
3             ICONST_M1 = 2;
4             ICONST_0 = 3;
5             ICONST_1 = 4; ...}
6
7  // define a BIR action for categories of byte codes
8  extension VM for ...VM {
9    actiondef zro (VM.F frame, int op);
10   actiondef one (VM.F frame, int op, int);
11   actiondef lcl (VM.F frame, int op, int localIndex);
12   actiondef fld (VM.F frame, int op, string fieldName);
13   actiondef typ <'a>(VM.F frame, int op);
14   actiondef cmp (VM.F frame, int op, lazy boolean result);
15   actiondef arr <'a>(VM.F frame, int dims);
16   actiondef inc (VM.F frame, int var, int inc);
17   ...
18 }
```

Figure 6. Bogor Java VM Extension Declaration

and the associated statespace. By using a domain-specific extension, the abstract semantics of the underlying event channel can be implemented directly in Java (hiding this complexity from the modeling level) and domain-specific optimizations for state-space search can be implemented (as explained in section 5).

Modeling machine instruction sets The Java Path Finder [7] model checker pioneered the approach of defining a model checker that works directly on the executable representation of a program. Using BIR's extension facilities, one can easily build a model checker for virtually any execution platform – each instruction for the platform (e.g., each Java byte code) is modeled directly as a new BIR action or expression. We have implemented complete model checkers for JVM byte code and .NET's CLR using this approach, and an undergraduate student at Brigham Young university used this strategy to implement a model checker for Motorola M68HC11 executables.

Figure 6 presents excerpts from a BIR extension for representing the Java VM instruction set. Line 2 begins a declaration that gives an integer tag for each JVM byte code. Then, BIR actions are declared for common categories of byte code. For example, one represents instructions with one integer parameter (e.g., `BIPUSH`), lcl represents instructions that access local slots (e.g., `ILOAD`), fld represents instruction that access object fields (e.g., `GETSTATIC`), typ represents instructions involving types (e.g., `CHECKCAST`), cmp represents comparison instructions (e.g., `IFEQ`), arr represents the `MULTINANEWARRAY` instruction, and inc represents increment/decrement instructions (e.g., `IINC`). Each action takes as parameters a frame that captures properties of the current execution

```
1  loc l0$27:
2      do {
3          VM.max(f, 3, 2); VM.set<(|Process1|)>(f, "this", 0);
4          VM.fld(f, Op.GETSTATIC, "/|Deadlock.state|\\");
5          } goto l1;
6    loc l1:
7      do {
8          VM.zro(f, Op.ICONST_1);
9          } goto l2;
10   loc l2:
11     do {
12         VM.zro(f, Op.IADD);
13         } goto l3$27;
14   loc l3$27:
15     do {
16         VM.fld(f, Op.PUTSTATIC, "/|Deadlock.state|\\");
17         } goto l4$28;
18   loc l4$28:
19     do {
20         VM.fld(f, Op.GETSTATIC, "/|Deadlock.lock1|\\");
21         } goto l5;
22   loc l5:
23     do {
24         VM.zro(f, Op.DUP);
25         } goto l6;
26   loc l6:
27     do {
28         VM.lcl(f, Op.ASTORE, 1);
29         } goto l7;
```

Figure 7. Bogor Java VM Extension Use (excerpts)

context, an integer tag representing the particular Java byte code being modeled, and additional arguments associated with the particular instruction category. Figure 7 presents excerpts of BIR model of bytecode program that uses the declared extension. This model excerpt results from a direct translation of a small Java program that contains a deadlock (Deadlock.state and Deadlock.lock1 are names of variables in that program).

5. Bogor's open architecture

Figure 8 presents the Bogor architecture. The architecture of Bogor can be divided into three parts: (1) a front-end that parses and type checks a given model expressed in the BIR modeling language, (2) interpretive components that implements the values the state transformations implied by BIR's semantics, and (3) model checking engine components that implement search and state storage strategies.

All model checking tools include functional aspects for state-space search, scheduling, and managing seen states. However, in their implementations, these aspects are often tangled, and thus insertion of alternate strategies or other customizations is often quite difficult. One of the contributions of our work is not just to present a non-tangled implementation, but moreover to present core components using widely-used and well-documented design patterns [23] that hide irrelevant implementation details by encapsulation, that reduce dependences between components, and that build in strategies for parameterization, adaptation, and extension.

For example, the `ISearcher` (implemented using the STRATEGY pattern) defines the search method used. For example, if depth-first search is used, then at any given state, its children states will be explored first before exploring its sibling states. The `IStateManager` (implemented using the FACADE pattern) provides an interface for storing states and for determining whether or not a state has been visited before. The `ISchedulingStrategist` (STRATEGY pattern) determines the scheduling strategy employed by the model checker. The most common strategy of model checkers is to generate all possible interleavings of thread executions. Other strategies include incorporation of support for priority based scheduling. In addition, when processing any inner-thread non-deterministic choice (e.g., associating multiply-enabled transitions within the same thread), this module should be consulted to determine which transition to execute next. For example, in a full-state exploration mode, the scheduler should make sure that every branch of a non-deterministic choice should be explored. This module is also consulted to determine which transformations are enabled in a given state.

More details about Bogor modules can be found in [39] and a complete listing of module APIs and examples of their use can be found on the Bogor web-site [43].

6. Kiasan – symbolic execution in Bogor

One of the most significant customizations of Bogor to date has been the development of Bogor/Kiasan[1] [16] – a Bogor-based symbolic execution framework for the JVM extensions of Figure 6. The research directions taken in Kiasan were driven by the fact that best practices in software development techniques heavily emphasize the development of reusable and modular software, which allows software components to be developed and maintained independently. One of the main challenges of component development is to ensure software compatibility across independently-developed components. Kiasan, which enables compositional verification of sequential Java programs, was developed to complement existing Bogor model checking technology which focuses on non-compositional verification of concurrent programs.

Kiasan was built by customizing the following Bogor modules.

- `IValueFactory`: Traditional *concrete* value representations are replaced by *symbolic* value representations. For values with composite structure (e.g., objects and arrays), this also involves creating the ability to form partial or incomplete composite representations that can be completed incrementally.

- `IStateFactory`: The conventional state representation that encodes global variables, program counters for threads, heap data, call stack frames, etc. is enhanced to hold symbolic data values and to include data constraints to represent path conditions.

- `IStateManager`: Kiasan's symbolic execution engine performs a stateless search. Thus, the traditional model checker's state storage and state matching implementations are changed to never store states and to always report that each state has not been encountered before.

- *JVM extension interpretation:* The extension code for intepreting the JVM byte code representations of Figure 6 is replaced by code that interprets byte codes symbolically – including exploration of both paths of conditionals (with accumulated path conditions) in cases where branch conditions cannot be decided.

Kiasan is similar to other frameworks that perform symbolic checking of Java programs (e.g., ESC/Java [21] and

[1] Kiasan (kē' ah sahn, Indonesian): to reason with analogy/symbolically

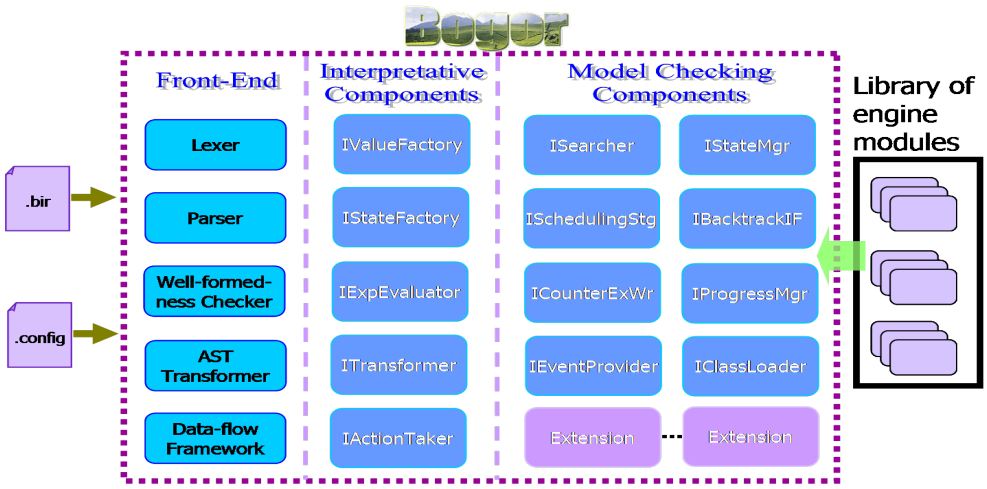

Figure 8. Architecture and primary modules of Bogor

the symbolic execution engine of JPF [34]) in that it only explores a bounded portion of a program's state space. However, Kiasan provides a number of useful capabilities that move beyond capabilities of related tools, including

- automatic checking of code against user-supplied contracts – pre/post-conditions, invariants, and assertions including specifications that capture *deep and strong properties of heap data*,
- compositional checking of methods/classes that leverages method contracts of invoked methods to summarize properties of invoked method implementations,
- ability to process implementations with partial or incomplete contracts by processing invoked method implementations directly instead of the contracts of those methods,
- flexible control over the depth/coverage of checking by adjusting tool parameters that bound the size of the heap structures considered by Kiasan,
- automatic generation of both abstract and concrete counterexample traces that indicate execution paths leading to detected errors, and
- automatic generation of JUnit test caes that guarantee complete path and heap coverage (within the resource bounds used to configure Kiasan).

Two key innovations enable these capabilities. The first innovation (building off [34]) is an improved strategy for lazy expansion of heap structures so that information about the heap is only instantiated when needed/touched. The second innovation of Kiasan is a flexible approach for stating the resource bounds that determine the explored state space. Specifically, the bounds are stated in terms of the length of

```
public class LinkedList<E> {
  class LinkedNode
  { E data;  LinkedNode next; }

  @NonNull LinkedNode head = new LinkedNode();

  //@ inv: isAcyclic();

  /*@ pre: isSorted(c)
    @      && other.isSorted(c);
    @ post: isSorted(c); @*/
  void merge(@NonNull LinkedList<E> other,
             @NonNull Comparator<E> c) {
    LinkedList<E> ll = new LinkedList<E>();
    LinkedNode n1 = this.head.next;
    LinkedNode n2 = other.head.next;

    while (n1 != null && n2 != null)
      if (c.compare(n1.data, n2.data) < 0) {
        ll.addLast(n1.data);
        n1 = n1.next; }
      else { ll.addLast(n2.data);
             n2 = n2.next; }
    while (n1 != null) {
      ll.addLast(n1.data);
      n1= n1.next; }
    while (n2 != null) {
      ll.addLast(n2.data);
      n2=n2.next; }
    head = ll.head;
  }
}
```

Figure 9. A Merge Example (excerpts)

possible heap reference chains. This enables users to gradually "dial up" the size/depth of the heap considered during program checking.

Kiasan works on a specification language with features similar to the Java Modeling Language (JML) [35], but we did not design our approach around a particular specification language. Instead, we only require a side-effect free specification that is transformable to an effective executable form. Existing methods such as `jmlc` for JML runtime monitoring [9] can be employed to transform specification into an executable form. In practice, one defines Kiasan

specification predicates as side-effect free Java methods.

To illustrate the capabilities and themes of Kiasan, consider the Java program in Figure 9. This program merges two sorted lists to obtain a resulting sorted list. Intuitively, the `merge` method's contract indicates that given a non-null and sorted (from the preconditions `@NonNull` and `pre`) acyclic list (from the invariant `inv`) with respect to the specified `Comparator c`, the method merges the content of that list into the receiver list object (given that it is also sorted) and as the result, the receiver object is also a sorted acyclic list (`isAcyclic` and `isSorted` are *pure* Java methods, i.e., they do not modify existing objects). We highlight several non-trivial challenges that Kiasan is able to overcome when reasoning about such programs and specifications.

1. The `compare` method is *open-ended*, *i.e.*, in contrast to other approaches that require a complete system such as [34] where the actual objects and the data being manipulated must be known, in this example the actual implementation of the `compare` method is unknown (and there may not even be an implementation for the type that will substitute `E`). Thus, the objects used to determine `compare`'s result are unknown as the method may use any data that it can reach. This example also poses greater challenges than programs considered by other approaches that reason about specific algorithms on data structures whose elements are of scalar types or (Java's) `String` [36, 47, 2]. In these other approaches, the restriction to simpler types allows one one to take advantage of the fact that the comparison operation does not use data from other heap objects. In our setting, the strongest specification we can have is that the `compare` method is pure, and it returns either a negative integer, zero, or a positive integer. Moreover, `compare` is a total order relation, as specified in the Java 5 Application Programming Interface (API) documentation.

2. The example also makes use of an `addLast` method (which we omit for lack of space). This method *only modifies* the last node's `next` field of the receiver list object by assigning its parameter to it. This is a case where it is actually easier to understand and use the implementation of a method than its specification. Since Kiasan can reason using the implementation of an invoked method instead of being restricted to using its contract, one can focus on checking `merge` first by using an implementation of `addLast`. Thus, Kiasan avoids one of the usability problems that occur when using pure compositional reasoning techniques such as [21] that require comprehensive specifications up-front before being able to check a program.

3. Consider strengthening the post-condition to ensure that (1) the resulting list size is the sum of the receiver list size before merging and the size of `other`, and (2) that all the elements are drawn from the two original lists. This property would be difficult to verify in other techniques that focus on properties of heap shapes [36], because these other techniques use heap abstraction techniques that "summarize" objects (which would likely not be able to keep track of specific properties of an unbounded number of elements in a list). This represents a trade-off between techniques: while other techniques can soundly prove properties for unbounded heap shapes, Kiasan can establish stronger properties within a bounded heap space (which, though unsound in general, is sound within the bounds given for a particular Kiasan session).

4. Verification of this method's contract requires proving that the information used for the comparison is not be modified by `merge` (or methods called from it); this will ensure the comparisons done later in `post` are unaffected.[2] If this fact cannot be established, there is no guarantee that the receiver object is sorted afterward. For example, suppose that we insert some code just before the end of `merge`. We need to check that the inserted code does not invalidate the elements' ordering. In this case, in would be enough to determine that the inserted code cannot does not modify the element objects. Kiasan is able to check this property by taking advantage of additional specifications that capture notions of heap region separation (e.g., specifications in this case would state the heap elements modified by the inserted code are disjoint from the list elements).

In summary, Kiasan is able to check the properties described above because its path-sensitive analysis based on resource-bounded (e.g., checking up to certain sizes of the two lists) symbolic execution is precise including its alias analysis. While it does not check for all possible sizes of the lists, however, it provides a strong behavior correctness guarantee up to the user-specified bounds. As faults are discovered and fixed, the user can opt to spend more computational resources to check larger bounds. The reader is referred to [16] for more thorough discussion on Kiasan.

7. Bogor within the development process

Bogor can be used as a stand-alone tool (via a command-line interface or an Eclipse-based graphical user interface) or encapsulated within other tools.

7.1. Eclipse-based graphical user interface

Bogor is implemented as a plug-in for *Eclipse* – a freely available open source and extensible tool platform from IBM. Eclipse provides a rich set of infrastructure for creating integrated development environments (IDEs) and graphical editors which makes it ideal for building a user interface (UI) for a model checking tool. Eclipse provides a plugin

[2] This is a bit too strong; it is fine to modify any information that will be accessed by `compare` if it does not change `compare`'s result.

facility by which one can add more functionality. The plugin facility of Eclipse matches well with Bogor's module extension facility. Eclipse, like Bogor, is implemented in Java. Thus, it allows a tight integration of Bogor and its GUI context and extensions.

Figure 7.1 gives screen shots from its Eclipse-based GUI. The first shot illustrates Bogor's counter-example navigation screen; the second illustrates Bogor's graphical display of the heap at a particular execution state.

7.2. Encapsulation in other tools

Our own experience of customizing Bogor with domain-specific modeling primitives and optimizations as well as encapsulating the resulting customized tool within larger environments has been quite positive.

We are developing the next generation of the Bandera [12] tools in Eclipse. This new version of Bandera analyzing models generated from Java source code using Bogor's rich built-in support for object-oriented language features. For these primitives, we have extended Bogor's default algorithms to support state-of-the-art model reduction/optimization techniques for object-oriented software using existing techniques such as collapse compression, heap symmetry, thread symmetry, and partial-order reductions. [18, 40].

For checking avionics system designs in Cadena, we customized Bogor's scheduling strategy to reflect the scheduling strategy of the real-time CORBA event channel and created a customized parallel state-space exploration algorithm that takes advantage of properties of periodic processing in avionics systems. These customizations for Bandera and Cadena have resulted in space and time improvements of over three orders of magnitude compared to our earlier approach [12, 26] which created models for SPIN and dSpin.

In other work, we have extended Bogor to support checking of specifications written in the Java Modeling Language (JML) [41, 45] and GUI frameworks [20].

7.3. Placement of model checking in a development process

Due to the computational costs associated with model checking, careful consideration must be given to the particular role it should play in a development methodology. In addition, one must keep in mind that the strength of model checking is reasoning about faults that arise due to non-determinism (e.g., the scheduling non-determinism that arises in concurrent software). There are many classes of errors that can be effectively detected using cheaper static analysis or testing techniques. Therefore, it seems reasonable to apply model checking *after* these cheaper techniques have been applied. An interesting open area for research is the development of techniques that allow coverage to be accumulated across multiple forms of quality assurance methods so that the effort of model checking can be pruned by considering coverage obtained by earlier testing or static analysis techniques.

Model checking at design time As we have already argued, it is natural to use model checking to reason about high-level behavioral designs (e.g., state-machine-like descriptions). The benefit of model checking at this level (as opposed to the implementation level) is that the system state-space is smaller since the design being checked represents an abstraction of the expected implementation.

Model checking implementations Even though techniques for model checking source code are maturing, in most cases it is impractical to consider using model checking for whole program analysis of large code bases. Many researchers believe that model checking can be successfully applied at the level of unit testing. In particular, unit testing for highly concurrent units seems to be a "sweet spot" for model checking.

One of the challenges in model checking software units is building an environment to close the system. When model checking is carried out in conjunction with unit testing, the required closing environment can often be based on the test harnesses used for unit testing.

Another challenge in model checking implementations is that most modern applications make extensive use of libraries. For Java programs, inclusion of all libraries associated with a program (based on the types used in the program) can result in a factor of 5-10 increase in the size of the code to be analyzed. This library code can either be abstracted or included along with the application program to form the model being analyzed. When the model checking framework works off of source/byte code and the libraries used include native methods, some abstract source/byte code representation *must* be developed for those methods. In recent work [17], we have shown that slicing techniques can be effective in pruning away library code that is irrelevant with respect to the particular property being model-checked.

Model checking mixed representations As software is increasingly built using models, frameworks, and large-scale infrastructure such as middleware, analysable system models will be built from a mix of implementations, higher level specifications and abstract models of infrastructure. For example, in [15] we presented a high-level framework for specification and implementation synthesis of synchronization aspects for system development. When model checking systems built from that framework, models

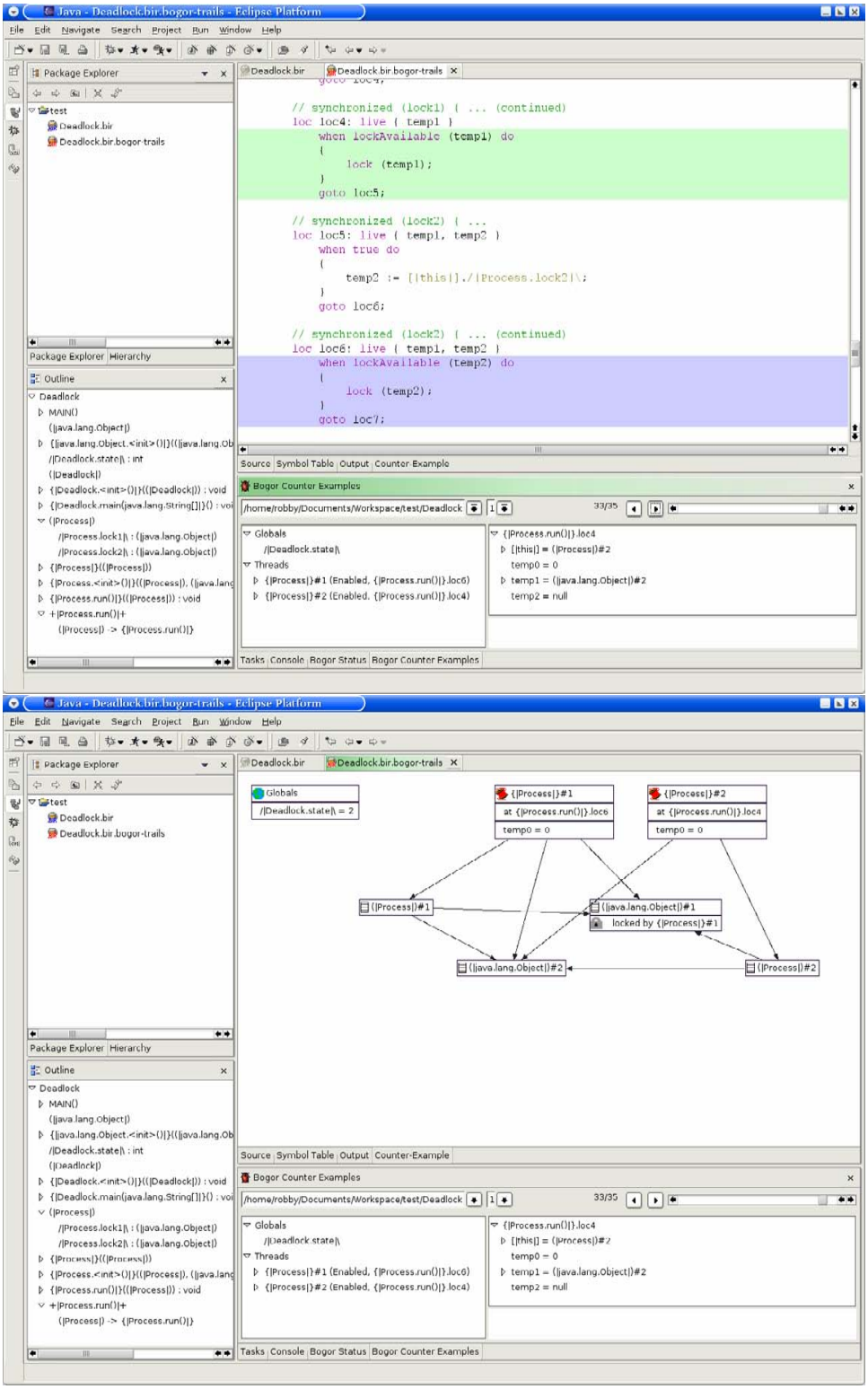

Figure 10. Bogor counterexample source code and heap structure displays

were constructed from joining the non-synchronization related implementation with high-level synchronization specifications (as opposed to the synthesized synchronization implementations). In this case, the high level descriptions were rich enough to completely determine the behavior of the synchronization aspects thus avoiding polluting the analyzed model with many irrelevant implementation details. Checking models built from high level synchronization descriptions resulted in multiple orders of magnitude reduction in checking costs compared to checking models built from synchronization implementations. We believe that this same strategy can be applied to other system aspects. In addition, when systems are built on reusable infrastructure, it may be worthwhile to handcraft abstract models of the infrastructure to reduce checking costs. This may be especially relevant in the context of software product-line development.

8. Pedagogical support for Bogor

The success and the expediency of the customization efforts described above were determined by several factors: (1) accessibility to domain knowledge, (2) intimate understanding of existing model checking algorithms, and (3) a model checking framework that allowed us to rapidly prototype ideas as concrete algorithms that we could experiment with. We believe that these factors influenced not only our specific experiences, but the experiences of others in applying Bogor as well. While issues in (1) should be addressed by the practitioners themselves, it is crucial for us to provide tutorial and reference material about Bogor's architecture and algorithms to enable others to successfully customize Bogor. Moreover, we believe Bogor itself is an excellent pedagogical vehicle for teaching foundations and applications of model checking because it allows students to see clean implementations of basic model checking algorithms and to easily enhance and extend these algorithms in course projects to include a variety of enhancements and optimizations.

Bogor comes with a user manual that includes BIR documentation (e.g., grammar, language description, abstract syntax tree implementation in Java, etc.) and Bogor extension tutorials that are directly accessible through the Eclipse help system as well as in PDF and HTML format. These materials refer to a well-documented repository of examples that illustrate how to construct various types of Bogor extensions.

To use Bogor as a pedagogical vehicle for teaching foundations and applications of model checking, we designed an extensive collection of freely available course materials for a one-semester course [44].

This course emphasizes a practical and project-oriented approach to learning the technical foundations of model checking and methodologies for applying model checking tools to realistic systems. Foundational topics covered include basic explicit-state reachability algorithms, temporal specification formalisms including LTL and CTL, partial order reductions, state-space representations (collapse compression, etc.), and alternate search strategies. In an approach similar to that used in compiler courses, these foundational and theoretical concepts are reinforced by having students implement key components of an explicit state model checker.

Students learn to apply Bogor to model and analyze simple concurrent systems that illustrate basic concepts of state-space exploration. Programming projects involve (re)implementing or modifying the core modules of Bogor's model checking engine, or implementing new modeling language primitives using Bogor's extensible modeling language. In addition to simply reinforcing the central concepts of model checking, the overall goal of these implementation exercises is to move students to the point where they can effectively develop model checking tools and associated methodologies for verification of real world systems by tailoring Bogor to different application domains.

Methodological aspects of model checking (and Bogor, in particular) are also emphasized. This includes repeatable strategies for capturing concurrent/distributed systems as effective verification models, applying abstraction and other state-space reducing model transformations, and using a pattern-based approach to constructing temporal specifications.

The course distribution for instructors includes a variety of pedagogical materials such as typeset lecture notes and guided exercises, PowerPoint lecture slides, streaming video for our lectures, source code for lecture examples, weekly quizzes and solutions, homeworks and solutions, exams and solutions. A separate distribution for students includes only the lecture slides and examples.

9. Summary of Experience Using Bogor

There is a growing user community for Bogor. In the past twelve months, Bogor has been downloaded more than 1000 times by individuals in 32 countries. We know that many of those individuals are using Bogor in interesting ways. To date, we are aware of more than 35 substantive extensions to Bogor that have been built by 28 people, only one of whom was the primary Bogor developer.

It is difficult to quantify the effort required to build a high-quality extension in Bogor. As with all software framework there is a learning curve. In the case of Bogor, which is a non-trivial system consisting more than 22 APIs, we find that reasonably experienced Java developers get up to speed in a couple of weeks. At that point extensions are generally require only a few hundred lines of code and often

they can be modeled closely after already existing extensions. To give a sense of the variety of extensions built with Bogor we list a sampling of those extensions and indicate, in parentheses, the number of non-comment source lines of Java code used to implement the extension.

Partial-order Reduction (POR) Extensions Multiple variations on POR techniques have been implemented in Bogor including: sleep sets (298), conditional stubborn sets (618), and ample sets (306) approaches. Multiple variations of the notion of dependence have been incorporated into these techniques that increase the size of the independence relation by exploiting : read-only data (515), patterns of locking (73), patterns of object ownership (69), and escape information (216). These latter reductions, while modest in size and complexity to implement, have resulted in more than four orders of magnitude reduction in model checking concurrent Java programs [18].

State-encoding and Search Extensions As explained earlier, Bogor is factored into separate modules that can be treated independently to help lower the cost of learning the framework's APIs. For example, extensions to the state-encoding and management APIs have yielded implementations of : collapse compression (483), heap and thread symmetry (317), and symmetric collection data structures (589). While extensions to Bogor's searcher APIs have enabled the POR extensions above in addition to one's supporting stateless search (14) and heuristic selective search (641) of the state space.

Property Extensions Supporting different property languages is just as important as supporting flexibility in modeling languages. Bogor's property APIs have allowed multiple checker extensions to be implemented including : regular expression/finite-state automata (1083), an automata-theoretic Linear Temporal Logic (1011) checker, and a Computation-tree Logic (1418) checker based on alternating tree automata. We have also implemented a checker extension for the Java Modeling Language [41] (3721).

Problem Domain Extensions A main objective of Bogor was to bring sophisticated state-space analyses to a range of systems and software engineering domains. Several extensions have been built that target specific issues in reasoning about multi-threaded Java programs, for example, treating dynamic class loading (425), reasoning about event-handler behavior in program written using the Swing framework [20], and reasoning about properties of method atomicity (359) [27]. In recent work, we have extended Bogor to support emerging approaches to reasoning about concurrency errors in program's written using the Message Passing Interface (358) [46].

In our work on the Cadena development environment [26] for designing component-based avionics systems, we have extended Bogor's modeling language to include APIs associated with the CORBA component model and an underlying real-time CORBA event service (2593). [14, 19]. For checking avionics system designs in Cadena, we have customized Bogor's scheduling strategy to reflect the scheduling strategy of the real-time CORBA event channel (439), and created a customized parallel state-space exploration algorithm that takes advantage of properties of periodic processing in avionics systems (516). These customizations for Bandera and Cadena have resulted in space and time improvements of over three orders of magnitude compared to our earlier approaches.

Other extensions of Bogor include checking and evaluating highly dynamic multi-agent systems [42]. Researchers outside of our group are extending Bogor to support checking of programs constructed using AspectJ, and UML designs, systems built on top of the Siena internet-scale publish/subscribe framework, and BPEL models.

10. Conclusion

In summary, we believe that a number of trends in software development suggest the need for flexible and adaptable infrastructure to enable practitioners to more effectively develop model checking tools that are customized to particular domains and development processes. While there are many avenues that one might pursue, we believe that the Bogor framework provides a robust and well-reasoned foundation that researchers and practitioners can use to address important facets of automated reasoning for modern software systems.

References

[1] R. Agarwal and S. D. Stoller. Type inference for parameterized race-free java. In *Proceedings of the Fifth International Conference on Verification, Model Checking and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 149–160. Springer-Verlag, Jan. 2004.

[2] S. Anand, C. S. Pasareanu, and W. Visser. Symbolic execution with abstract subsumption checking. *Proceedings of 13th International SPIN Workshop on Model Checking of Software (SPIN2006)*, 2006. (to appear).

[3] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI-01)*, pages 203–213, June 2001.

[4] T. Ball and S. Rajamani. Bebop: a symbolic model-checker for boolean programs. In K. Havelund, editor, *Proceedings of Seventh International SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*, pages 113–130. Springer-Verlag, 2000.

[5] S. Ben-David, C. Eisner, D. Geist, and Y. Wolfsthal. Model checking at ibm. *Formal Methods in System Design*, 22(2):101–108, 2003.

[6] D. Bosnacki, D. Dams, and L. Holenderski. Symmetric SPIN. *International Journal on Software Tools for Technology Transfer*, 4(1):92–106, 2002.

[7] G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder – a second generation of a Java model-checker. In *Proceedings of the Workshop on Advances in Verification*, July 2000.

[8] W. Chan, R. J. Anderson, P. Beame, D. Notkin, D. H. Jones, and W. E. Warner. Optimizing symbolic model checking for statecharts. *IEEE Transactions on Software Engineering*, 27(2):170–190, 2001.

[9] Y. Cheon and G. T. Leavens. A runtime assertion checker for the Java modeling language. *Software Engineering Research and Practice*, pages 322–328, 2002.

[10] 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.

[11] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[12] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera : Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 439–448, June 2000.

[13] C. Demartini, R. Iosif, and R. Sisto. dSPIN : A dynamic extension of SPIN. In *Theoretical and Applied Aspects of SPIN Model Checking (LNCS 1680)*, Sept. 1999.

[14] W. Deng, M. Dwyer, J. Hatcliff, G. Jung, and Robby. Model-checking middleware-based event-driven real-time embedded software. In *Proceedings of the 1st Internatiuonal Symposium on Formal Methods for Component and Objects*, pages 154–181, 2002.

[15] X. Deng, M. B. Dwyer, J. Hatcliff, and M. Mizuno. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *Proceedings of the 24th International Conference on Software Engineering*, pages 442–452, 2002.

[16] X. Deng, J. Lee, and Robby. Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *Proceedings of the 21st IEEE Conference on Automated Software Engineering*, Nov. 2006. (to appear).

[17] M. B. Dwyer, J. Hatcliff, M. Hoosier, V. P. Ranganath, Robby, and T. Wallentine. Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. In *Proceedings of the Tweflth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2006)*, Lecture Notes in Computer Science, Mar. 2006.

[18] M. B. Dwyer, J. Hatcliff, Robby, and V. R.Prasad. Exploiting object escape and locking information in partial order reduction for concurrent object-oriented programs. *Formal Methods in System Design*, 25(2-3):199–240, 2004.

[19] M. B. Dwyer, Robby, X. Deng, and J. Hatcliff. Space reductions for model checking quasi-cyclic systems. In *Proceedings of the Third International Conference on Embedded Software*, 2003.

[20] M. B. Dwyer, Robby, O. Tkachuk, and W. Visser. Analyzing interaction orderings with model checking. In *Proceedings of the 19th IEEE Conference on Automated Software Engineering*, 2004. (to appear).

[21] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. *Programming Language Design and Implementation*, 2002.

[22] FormalSystems. FDR2 website. `http://www.fsel.com/`, 2003.

[23] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley Pub. Co., 1995.

[24] D. Garlan, S. Khersonsky, and J. S. Kim. Model checking publish-subscribe systems. In *Proceedings of the 10th International SPIN Workshop on Model Checking of Software*, pages 166–180, May 2003.

[25] P. 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, Jan. 1997.

[26] J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. Prasad. Cadena: An integrated development, analysis, and verification environment for component-based systems. In *Proceedings of the 25th International Conference on Software Engineering*, pages 160–173, 2003.

[27] J. Hatcliff, Robby, and M. B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model checking. In M. Young, editor, *Proceedings of the Fifth International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2004)*, volume 2937 of *Lecture Notes In Computer Science*, pages 175–190, Jan 2004.

[28] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, pages 58–70, Jan. 2002.

[29] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.

[30] R. Iosif. Symmetry reduction criteria for software model checking. In *Proceedings of Ninth International SPIN Workshop*, volume 2318 of *Lecture Notes in Computer Science*, pages 22–41. Springer-Verlag, Apr. 2002.

[31] R. Iosif, M. B. Dwyer, and J. Hatcliff. Translating Java for multiple model checkers: the Bandera back end. *International Journal on Formal Methods in System Design*, 2004. (to appear).

[32] F. Ivancic, I. Shlyakhter, A. Gupta, M. Ganai, V. Kahlon, C. Wang, and Z. Yang. Model checking c programs using f-soft. In *Proceedings of the 2005 International Conference on Computer Design (ICCD 2005)*, pages 297–308, Oct. 2005.

[33] C. T. Karamanolis, D. Giannakopolou, J. Magee, and S. M. Wheather. Model checking of workflow schemas. In *4th International Enterprise Distributed Object Computing Conference*, pages 170–181, Sept. 2000.

[34] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. *Tools and Algorithms for Construction and Analysis of Systems*, pages 553–568, 2003.

[35] G. T. Leavens, A. L. Baker, and C. Ruby. JML: a Java modeling language. In *Formal Underpinnings of Java*, 1998.

[36] T. Lev-Ami and M. Sagiv. Tvla: A framework for kleene-based static analysis. In *Proceedings of the 7th International Static Analysis Symposium (SAS'00)*, 2000.

[37] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1991.

[38] L. Nachmanson, M. Veanes, W. Schulte, N. Tillmann, and W. Grieskamp. Optimal strategies for testing nondeterministic systems. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 55–64, New York, NY, USA, 2004. ACM Press.

[39] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *Proceedings of the 9th European Software Engineering Conference held jointly with the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 267–276, 2003.

[40] Robby, M. B. Dwyer, J. Hatcliff, and R. Iosif. Space-reduction strategies for model checking dynamic software. In *Proceedings of the 2nd Workshop on Software Model Chekcing*, volume 89(3) of *Electronic Notes in Theoritical Computer Science*, 2003.

[41] Robby, E. Rodríguez, M. B. Dwyer, and J. Hatcliff. Checking strong specifications using an extensible software model checking framework. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 404–420, Mar. 2004.

[42] Robby, Scott A. DeLoach, and Valeriy Kolesnikov. Using Design Metrics for Predicting System Flexibility. In *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 26 - April 2, 2006, Proceedings*, volume 3922 of *Lecture Notes in Computer Science*, pages 184–198. Springer, 2006.

[43] SAnToS Laboratory. Bogor Website. `http://bogor.projects.cis.ksu.edu`, 2003.

[44] SAnToS Laboratory. Software Model Checking course materials website. `http://model-checking.courses.projects.cis.ksu.edu`, 2003.

[45] SAnToS Laboratory. JML-Eclipse Website. `http://jmleclipse.projects.cis.ksu.edu`, 2004.

[46] S. F. Siegel. Efficient verification of halting properties for mpi programs with wildcard receives. In *Verification, Model Checking, and Abstract Interpretation: 6th International Conference (VMCAI 2005)*, volume 3385 of *Lecture Notes in Computer Science*, Jan. 2005.

[47] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2005.