EXPLORING DOMAIN SPECIFIC APPROACHES TO SOFTWARE MODEL CHECKING

By

MINAL V. WAD

Bachelor of Science

Mumbai University

Maharashtra, India

2004

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December 2006

EXPLORING DOMAIN SPECIFIC APPROACHES TO

SOFTWARE MODEL CHECKING

Thesis Approved:

Dr. M. H. Samadzadeh

Thesis Adviser

Dr. J. P. Chandler

Dr. N. Park

Dr. Gordon Emslie

Dean of the Graduate College

PREFACE

Model checking has proven to be an effective technology for verification and debugging in hardware domains and more recently in software domains. The major challenges in the application of model checking to software systems are: the mapping of software executables to model checker's input language and the intrinsic complexity of the ever growing software systems. This thesis explores the domain specific model checking approaches to large systems in order to optimize the state space storage for specific domains.

Bogor [Bogor 2003] is an extensible, customizable, and highly modular model checking framework that supports general as well as domain specific software model checking. As a part of the thesis, domain specific extensions to Bogor's input language, called Bandera Intermediate Representation (BIR), were implemented by providing a plugin for Eclipse [Eclipse 2004]. Eclipse is a universal platform for tool integration and its plugin development environment facilitates addition of new plugins to the existing ones. Eclipse's extension mechanism is exploited by Bogor. Bogor was installed as an Eclipse plugin and with the help of Eclipse's Plugin Development Environment (PDE), new data types were integrated with the existing Bogor framework.

Two case studies ('postfix calculator' using stack extension and 'resource allocation' using multiset extension) were investigated. Various metrics such as number

of states, transitions, and maximum depth were analyzed. The complexity of the test cases was increased gradually to test the extensions for feasibility and scalability. The thesis also involves a comprehensive study of some of the well-known model checkers and their features, degree of automation, and input languages.

It was observed that customizing the model checker as per domain specifications helped in achieving space reduction. The space reduction is prominent, especially in large domains where it contributes towards state space explosion solution. Although development of extensions is achievable, it requires a working knowledge of Eclipse and specific knowledge of model checking. In conclusion, a domain specific approach for software model checking was demonstrated to be a promising technology. Language extensions to BIR were successfully built and tested for accuracy and scalability.

ACKNOWLEDGEMENTS

I would like to express my gratitude to all those who gave me the hope and encouragement to complete this thesis. I would like to extend my sincere appreciation to Dr. M. H. Samadzadeh for being a perfect mentor, assisting me with his wisdom, encouragement, and patience throughout my graduate studies at Oklahoma State University. My special gratitude is also extended to Dr. N. Park and Dr. J. Chandler for their valuable help while serving on my committee.

My appreciation goes to the soul of my late grandmother whose blessings accompany me wherever I go. My special thanks are due to my parents and brother Charudatta for their love and support. I would like to extend my special gratitude to my fiancé, Siddharth for being patient and understanding throughout my research.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

INTRODUCTION

1.1 Importance of Software Model Checking

There is hardly any aspect of our day to day life where software does not play a crucial role. The increasing involvement of software in safety critical systems has made it imperative to validate them rigorously before deployment. With the advent of a formal verification method called model checking [Clarke et al. 1999], the process of validation is made exhaustive and automatic to some extent. Model checking has been successfully used in the past for validation and debugging in hardware [Clarke and Mishra 1983] and more recently in software domains. Due to the escalating complexity in software artifacts, the growth of software model checking industry has been somewhat hampered. Despite intensive research on general techniques to reduce the complexity of model checking, state space explosion and scalability remain the major obstacles to its adoption. While general reduction strategies are employed to enable space reductions [Schuppan and Biere 2004], it has been observed that, by applying explicit knowledge of a domain, one can replace large portions of a state space with smaller structures, thus allowing greater degree of state space reduction [Hoosier et al. 2004].

1.2 Domain Specific Approach to Software Model Checking

Domain specific approaches to model checking represent the model with fewer variables, thus potentially reducing the size of the state space. Experts in different software domains have significant knowledge about the semantics and properties of their respective domains. Cost effective domain specific model checkers can be built with the help of such domain experts. Domain specific models can be built from scratch [Brat et al. 2000] [Godefroid 1997] or by instantiating and targeting the existing extensible model checkers [Chan et al. 2001] [Demartini et al. 1999]. This thesis discusses, studies, and uses Bogor [Bogor 2003] which is a novel model checker that supports customization and extensibility and is easily embedded or encapsulated in larger development tools.

1.3 Related Work and the Scope of Thesis

Domain customization is seen in real-time applications [Hoosier et al. 2004], where Bogor is customized for checking properties of avionics design. New Bogor ADT (Abstract Data Type), new scheduler, new state vector representation and new Bogor internal modules were developed to capture the real-time behavior. Bogor architecture can be modified [Robby et al. 2004] to check JML specifications of sequential and concurrent Java programs. A similar approach can be used to tailor the Bogor model checker [Robby et al. 2006] to efficiently analyze the adaptive behaviors of multiagent systems [DeLoach and Matson 2004] and to determine their properties such as flexibility, fault-tolerance, and cost-efficiency. Bogor tutorial on developing extensions [Dwyer et al. 2005] explores extension development and provides pedagogical material useful for customizing Bogor's input language.

This thesis work advocates domain specific approaches for software model checking in order to enhance memory reduction. Reduction of both the number of program states that are stored and the size of those states are presented. Empirical data supporting the effectiveness of these memory reductions on a collection of realistic examples is presented. Domain customization is achieved by providing extension to Bogor's input language BIR (Bandera Intermediate Representation) using Eclipse's PDE (Plugin Development Environment).

CHAPTER II

REVIEW OF LITERATURE

2.1 Model Checking - Background

Model checking is an automatic technique for verifying finite state concurrent systems [Clarke et al. 1999]. In this approach the system to be verified is represented as a finite state transition system [Zohar and Amir 1992] and the properties are expressed in temporal logic [Eleftherakis and Kefalas 2001]. Using model checking, one can determine if a given system satisfies the required specification and behaves appropriately in a given circumstance.

Not only is model checking largely automatic and comprehensive, but it also generates useful feedback in the form of counter-examples. The counter-examples describe the states of a system at every significant transition. By following the hints, a user can debug a faulty program or can just manipulate the specifications if the logic of the program seems to be correct. By iterating the verification process, the sources of the errors can be located without using traditional testing methods or theorem proving principles. Also, since model checking is comprehensive, most of the potential behaviors are tested, thus reducing the probability of inadequate or missed behavior.

In the past couple of decades since model checking has emerged, there has been a lot of research on software model checking. Various kinds of temporal logics [Emerson 1990] have been extensively studied and efficient model checking algorithms [Clarke et al. 1986] [Queille and Sifakis 1982] have been designed.

The model checking process consists of three steps: modeling, specification, and verification. The output of modeling combined with the output of specification is given as input to verification. Verification is generally an iterative phase; it is repeated till a desired model is obtained.

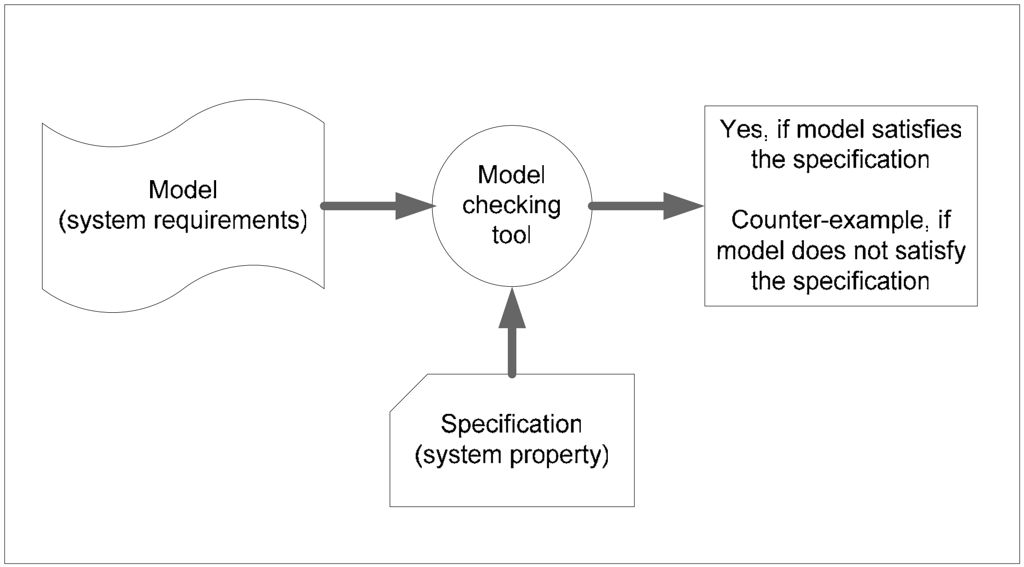


Figure 1. The Process of Model Checking

2.1.1 Kripke Structure

A Kripke Structure can be viewed as a transition diagram that captures the intuition about the behavior of reactive systems. A Kripke Structure consists of a finite set of initial states, a set of transitions between states, and a function that labels each state with a set of properties that are true in that state [Clarke et al. 1999]. A Kripke Structure is often represented as a forest of computation trees. For each initial state, a computation tree can be constructed by unraveling the Kripke Structure into an infinite tree. Figures 2 and 3 [Clarke et al. 1999] show the Kripke Structure of a system and the corresponding computation tree that is obtained by unwinding the Kripke Structure.

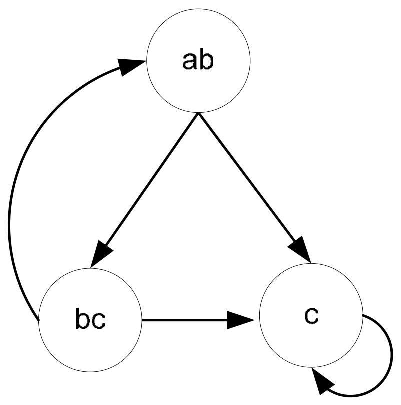


Figure 2. A Kripke Structure

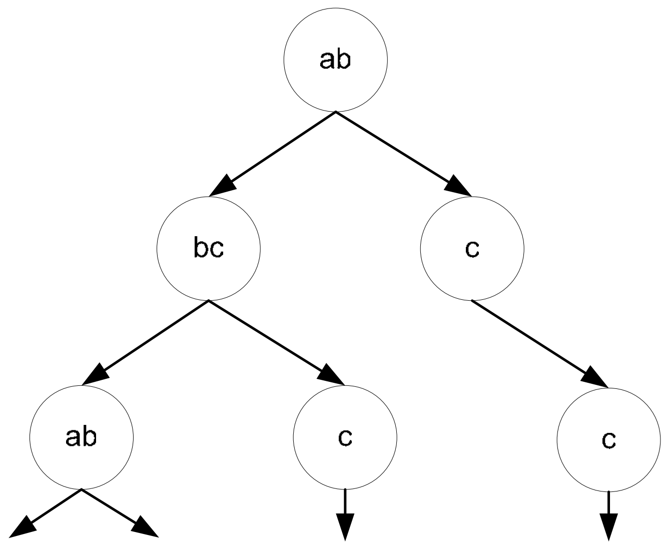


Figure 3. A Computation Tree

Each state in a Kripke Structure essentially contains one value for each state variable. A transition denotes a change in the value of one or more state variables. A Kripke Structure is unfolded and converted into an infinite tree, where each path in the tree indicates a possible execution or behavior of the system.

Let AP be the set of Atomic propositions. A Kripke Structure M over AP is a four tuple $M = (S, S_o, R, L)$ where

S is a finite set of states,

$S_{o,}$ a subset of S, is the set of initial states,

R, a subset of S × S, is a transition relation that must be total, that is, for every state s in S there is a state s' in S such that (s, s') is in R, and

L: S → 2^{AP} is a function that labels each state with the set of atomic propositions true in that state.

2.1.2 Temporal Logics

The properties of state transition systems or Kripke Structures are described using temporal logic. Temporal logic is associated with time along with atomic properties. Temporal logic makes use of Boolean connectives such as conjunction, disjunction, and negation to describe the properties of a system. Although time is associated with temporal logic, the relationship is not explicit. In temporal logic, qualifiers such as 'eventually' and 'never' are used to describe time constraints associated with predicates.

CTL* (Computational Tree Logic) is a logic that combines both branching-time and linear-time operators. CTL* formulas are composed of path quantifier and temporal operators. CTL* is described by using computation trees. A computation tree is a tree, with an initial state as its root, which is then unwound into an infinite tree. All possible executions are covered in a computation tree.

2.2 State Space Explosion

One of the problems with model checking is state space explosion. This problem occurs in systems with many components that can interact with each other, or in systems with data structures that can assume many different values. In such cases, the number of global states can be enormous. If a system consists of many components that can make transitions in parallel, then it is difficult to verify such a system. In such systems, the number of global states may grow exponentially with the number of processes [Clarke et al. 1999]. Such systems pose the problem of state space explosion. However, researchers have come up with techniques based on Automata Theory and Symbolic Structures [McMillan 1992] to reduce the size of transition systems.

2.3 Trends in Software Model Checking

Traditional methods, e.g., simulation and testing, have long been in use for software verification and validation [Visser et al. 2000]. While relatively successful, these traditional methods become increasingly difficult to utilize as the complexity of software increases. Interpreting the interleaving and the control flow of concurrent programs and the debugging of multithreaded programs is non-trivial. The computing world, thus felt a need for automatic verification of programs.

Software model checking is generally difficult to implement because of the potentially enormous state space. Recently, considerable progress has been made to mitigate the problem of state explosion. Some of the successful techniques that deal with the problem of state space explosion are partial order reduction [Godefroid and Pirottin 1993], symbolic model checking [Burch et al. 1994], bounded model checking [Biere et al. 1999], compositional reasoning [Clarke et al. 1989], and abstraction [Clarke et al. 1994].

A number of model checkers are readily available and some of them are in widespread use. Most of these existing model checkers are designed to support a predetermined input language and a fixed platform. Some of the well-known model checkers are SPIN, NuSMV, Bandera, BLAST, and Verisoft. Part of this thesis work involved a detailed study of the widely known model checkers based on their features, degree of automation, input language, use of GUI, and working platform.

2.4 Drawbacks of Most Existing Model Checkers

Despite an appreciable amount of progress, most of the existing model checkers exhibit the following deficiencies.

1. Inadequate Mapping of a System into a Model Checker's Input Language: Modeling a system, which is mostly represented in the form of code segments or state diagrams, into a formalism that is understood by a model checker, is a significant step in the process of model checking. Generally, a model checker's input language has a fixed set of syntax rules and constructs that act as restrictions for replicating the exact behavior of a system. The stereotyped behavior of most of the input languages causes complications and may result in accurate representation of concurrent systems or systems involving dynamic creation of states.
2. Standardized Architecture: Most of the model checkers have a fixed pattern for state encodings, search algorithms, and reduction strategies [Robby et al. 2003b]. Each system has a collection of components, properties, and data pertaining to its domain. A domain specific approach leverages a scope for efficient encoding of system states, and further enhances the state space reductions without incurring unnecessary

overhead. Depending on the nature of the system, it is desirable to configure the search algorithms such as a stateless search [Godefroid 1997] or a less heuristic search for defect detection [Edelkamp et al. 2001] [Groce and Visser 2002].

Thus, there is a need for model checking tools that support customization and extensibility and that are easily embedded or encapsulated in larger development tools.

CHAPTER III

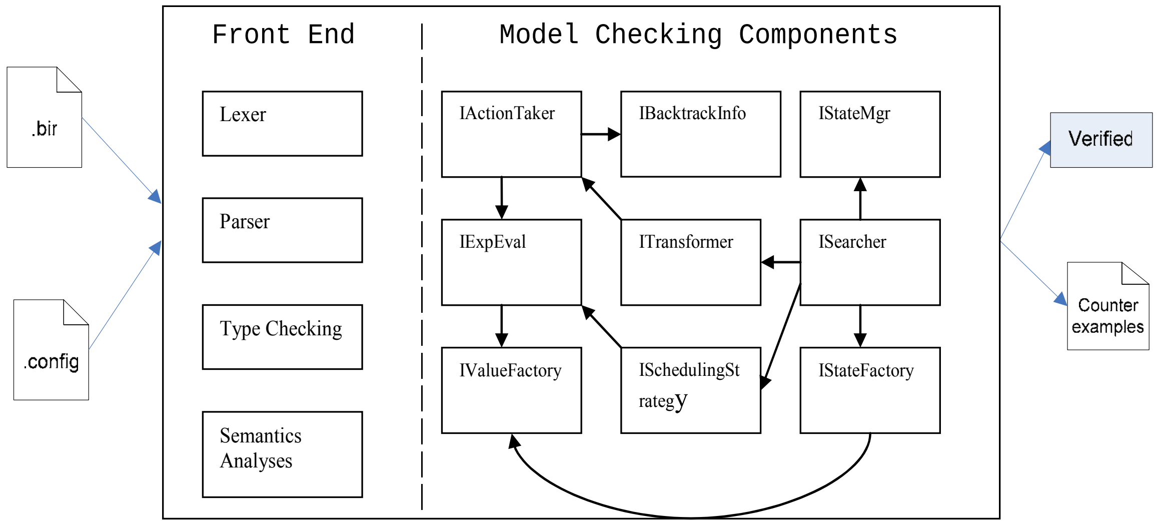
BOGOR AND ECLIPSE

3.1 Bogor Model Checker

Bogor [Bogor 2003] is a novel model checking framework that has been developed by the SAnToS group (Specification, Analysis, and Transformation of Software) at Kansas State University and the ESQuaReD group (laboratory for Empirically-based Software Quality Research and Development) at University of Nebraska, Lincoln. Bogor is designed to support the analysis of a wide variety of software artifacts related to modern, dynamic, and concurrent software systems. The extensible framework of Bogor allows domain experts, who are not necessarily model checking experts, apply model checking without the need to build their own model checkers or having to pour over the details of an existing model checker implementation while carrying out substantial modifications.

Figure 4 below shows the architecture of Bogor. The architecture is divided into two parts: Front End and Model Checking Components. Bogor has a modular architecture with nine prime bogor components each of which is implemented using a plugin. The arrowheads show the interdependencies between modules, e.g., the IActionTaker model uses the IBacktrackingInfo module.

The front end builds the abstract syntax tree (AST) from the input model, checks the well-formedness of the model by type checking, and performs extensions interface checking. Bogor model checker components contain three major modules: i) Search Module, ii) Scheduling Module, and iii) State Manager Module, as well as other modules that manage backtracking and extensions [Robby et al. 2003b].



(.config and .bir are the inputs to the model checker consisting of the configuration and intermitiate representation files)

Figure 4: Bogor Architecture [Robby et al. 2003b]

3.1.1 Bogor Model Checking Framework

Bogor [Bogor 2003] is an extensible software model checking framework that has been designed to support general purpose model checking as well as domain specific software model checking. Bogor's model checking algorithms, user interface, and other features apparently make it more adaptable than most of the existing model checkers. Some of the features of Bogor are listed below.

- Modeling Language Supporting Object-Oriented Features: Bandera Intermediate Representation language (BIR) not only provides basic constructs (that are normally found in most modern programming languages) but also includes the dynamic creation of objects and threads, garbage collection, and exception handling [Dwyer et al. 2005]. This bridges the gap between translation of a software artifact that uses a modern, modular, and object-oriented methodology, into a Bogor compatible model.
- An Extensible Modeling Language: Bogor's modeling language, BIR, can be extended to a particular domain (such as multi-agent systems, avionics, or security protocols) or with respect to a particular level of abstraction (such as design models, source code, or byte code). BIR can include new primitive types, expressions, and commands [Dwyer et al. 2005].
- Open Modular Architecture That Facilitates Encapsulation: Bogor's well-organized modular structure allows Bogor's default model checking algorithms to be replaced by new algorithms and new optimizations. Bogor has adapted and extended the optimization/reduction strategies such as heap and thread symmetry [Robby et al. 2003a], collapse compression [Holzmann 1997], and partial order reductions. Bogor's open architecture extends its encapsulation into a domain specific environment with less difficulty than most of the existing model checkers.
- Robust Graphical Interface: The feature-rich user interface of Eclipse provides a variety of visualization and navigation facilities [Dwyer et al. 2005]. The robustness of Eclipse's GUI can be utilized by installing Bogor as a plugin for Eclipse.

3.1.2 BIR Modeling Language

Bandera Intermediate Representation (BIR) is a modeling language of the Bogor model checker [Robby et al. 2003b]. BIR supports the object-oriented paradigm, hence it could be considered a pragmatic modeling language for expressing concurrent systems.

BIR's support for modeling software artifacts ranges from languages such as Java and C#, and design levels such as transition diagrams and state charts, to abstractions of software layers as in Common Object Request Broker Architecture (CORBA) middleware services [Hoosier et al. 2004] and communication mechanisms.

BIR's primitive types include boolean, integer, subranges, and enumerated types and its non-primitive types include record, array, and lock. BIR also provides support for abstract data types (ADT), polymorphic functions, and dynamic creation of both thread and heap objects [Robby et al. 2003b].

Figure 5 below represents BIR model of two concurrent processes. TwoDiningPhilosopher.bir is an implementation of the classic problem of 5 Dining philosophers. In this example, two philosophers think and eat without doing any talking. There is a bowl of spaghetti in the center of the table along with and two plates and two forks. Each philosopher requires 2 forks to eat. Philosopher 1 (P1) would like to grab fork1 (right fork) and then fork2 (left fork), and start eating. On the other end, philosopher 2 (P2) would like to lift fork 2 then fork 1 (in that order), and start eating. There is a possibility that this system could reach a deadlock if both philosophers are holding a fork in one hand and waiting on the other philosopher to free the other fork. For each philosopher there is a thread created in the BIR model as shown below in Figure 5.

```
system TwoDiningPhilosophers
{

  boolean fork1 := false;
  boolean fork2 := false;

  active thread Philosopher1()
  {
    loc loc0: live {} // take first fork
      when !fork1 do { fork1 := true; }
      goto loc1;

    loc loc1: live {} // take second fork
      when !fork2 do { fork2 := true; }
      goto loc2;

    loc loc2: live {} // put second fork down
      do { fork2 := false; }
      goto loc3;

    loc loc3: live {} // put first fork down
      do { fork1 := false; }
      goto loc0;
  }

  active thread Philosopher2()
  {
    loc loc0: live {} // take second fork
      when !fork2 do { fork2 := true; }
      goto loc1;

    loc loc1: live {} // take first fork
      when !fork1 do { fork1 := true; }
      goto loc2;

    loc loc2: live {} // put first fork down
      do { fork1 := false; }
      goto loc3;

    loc loc3: live {} // put second fork down
      do { fork2 := false; }
      goto loc0;
  }

}
```

Figure 5. BIR Example [Bogor 2003]

3.1.3 Customizing Bogor to a Domain

Bogor has an extensible architecture that allows adding new data types to BIR and swapping the existing search strategies with new domain specific strategies. This feature can be exploited to build a customized model checker. It turns out that often times there are certain components of a software system that could have a significant number of states that are irrelevant to the properties being checked. By introducing new and encapsulated BIR native data types, together with operations to manipulate them, the code complexity can be pushed into the model checker's runtime environment instead of into the model itself.

Extensions do not change or extend the grammar of BIR, hence the built-in parser or syntactic symbols need not be changed as is the case in some other model checkers such as SPIN [Holzmann 1997] that require parser modification. Extensions make use of the already existing Bogor model checker components and the newly developed Java packages and classes. Bogor recognizes new extensions by a block of extension definition code as shown in Figure 6.

```
extension extension_name for MyPackage.MyModule
{
// Type definition
typedef type<'a>
// Action definition and expression definition
functions
}
```

Figure 6. An Extension Definition

The extension definition code has the keyword ‘extension’ followed by the name of the extension, followed by the name of the java class that implements the action definition and expression definition functions.

The idea of extension development is to let the Java package (i.e., the one that implements the extension) hold the state associated with the complex component and only expose as much of it as is relevant at the BIR level, rather than maintaining a complete implementation of a software component using BIR's variables [Bogor 2003]. The plugin development environment (PDE) of Eclipse [Eclipse 2004] can be used to provide extensions to Bogor’s input language BIR.

3.2 Using Eclipse for Simulation

Eclipse is a Java Integrated Developing Environment (IDE). Its framework can be extended by a developer to integrate new functions [Eclipse 2004]. Eclipse provides a plugin facility via which one can add more features. Bogor is implemented as an Eclipse plugin and, using the plugin development environment (PDE) of Eclipse, one can implement extensions and replacement strategies for Bogor modules. Eclipse’s extension point mechanism, which allows new plugins to contribute functionality to the existing ones, helps in testing, debugging, and running of the extensions before they are deployed.

CHAPTER IV

CASE STUDY, RESULT, AND COMPARISON

This chapter describes the creation and implementation of BIR language extensions Stack and Multiset. These extensions were developed using the plugin development environment (PDE) of Eclipse 3.0 and were debugged in the Eclipse's extension point environment by instantiating a new Eclipse environment. Two BIR models (postfix calculator and resource contention) were used as test cases to test different operations on the abstract data type extensions. Furthermore, both examples were tested for accuracy, operational capability, generation of counter example, etc., to show their functional potential as well the working of the Bogor model checker.

Sections 4.1 and 4.2 cover two specific cases describing data types, design, basic operations, implementation of the extension, and excerpts of code. Subsections 4.1.3, 4.1.4, 4.2.3, and 4.2.4 describe the observations and Bogor trail files for both case studies, and they also cover the results and a discussion of the results.

4.1 Case Study - Postfix Calculator (Stack Extension)

Postfix calculators employ reverse polish notation. To evaluate an expression in postfix, we need to express it without any parenthesis or precedence, e.g., ((3 + 4) * 5) + 1 will be expressed as 3 4 + 5 * 1 +

The expression is then evaluated from left to right using a stack by going through the following steps.

- push when encountering an operand
- pop two operands and evaluate the value when encountering an operation
- push the result

Thus the abstract data type ‘Stack’ is most suited for postfix evaluation. Stack is not a native BIR construct, though it can be implemented using BIR Arrays. If the code is to be written using arrays, then the coder has to write an extra piece of code to make an array behave like a stack, i.e., in LIFO fashion, checking for emptiness, etc. Basic operations such as PUSH ( ), POP ( ), and isEmpty ( ) are to be implemented using functions.

Two models of the Postfix Calculator were developed: one model uses the classic approach of using arrays as stack, whereas the other model adopts an extension mechanism by using the ‘Stack’ extension to Bogor. The Stack extension was developed using Eclipse’s PDE and the extensible architecture of Bogor. The methodology of extension development and implementation of the postfix calculator using the stack extension are described below.

4.1.1 Extension Syntax

In order to implement the stack extension, the following operations on stack were required to be defined.

List of Operations:

▪ Create ( )	- Create a stack with the given data type for the elements
▪ Push ( )	- Push an element on top of the stack
▪ Pop ( )	- Pop the top element of the stack
▪ IsEmpty ( )	- Return a boolean
▪ getTop ( )	- Get the element at the top of the stack without popping it
▪ Size ( )	- Return the size of the stack

The following code segment informs the Bogor language recognizer about the new extension type. The keyword ‘extension’ is followed by extension name ‘Stack’, followed by java class ‘myStackModule’. The type of this extension is kept generic and can be instantiated to any data type at the time of declaration. (e.g., Stack.type<int> MyStack; declares an integer stack).

```
extension Stack for bogor.MyStack.myStackModule
      {
            typedef type<'a>;

            expdef boolean isEmpty<'a>(Stack.type<'a>);

            expdef Stack.type <'a> create <'a> ('a);

            expdef int size<'a>(Stack.type<'a>);

            expdef 'a getTop<'a>(Stack.type<'a>);

            actiondef push<'a>(Stack.type<'a>, 'a);

            actiondef pop<'a>(Stack.type<'a>);
      }
```

Figure 7. Extension Definition for Stack

4.1.2 Implementation Semantics

Once the extension definition is constructed, the focus is shifted to the java class that actually implements the functionality of the extension. Every language extension java class has to implement the IModule interface that is provided by the Bogor framework [Bogor 2003] along with its required methods (see Figure 8 below).

The Connect ( ) method establishes connection with the main Bogor model checking components. The getCopyrightNotice ( ) and setOptions ( ) methods are sometimes used to display legal messages and configure advanced options. Along with the required methods, java class 'myStackModule.java' also implements each of the operations (create, push, pop, etc.) as stated in the extension definition.

```
package bogor.MyStack;

public class myStackModule implements IModule
{

      public IMessageStore connect(IBogorConfiguration bc)
      {
        tf = bc.getSymbolTable().getTypeFactory();
        ee = bc.getExpEvaluator();
        ss = bc.getSchedulingStrategist();
        vf = bc.getValueFactory();
        bf = bc.getBacktrackingInfoFactory();
        return new DefaultMessageStore();
      }

      public String getCopyrightNotice()
      {
            return null;
      }

      public IMessageStore setOptions(String arg0, Properties
arg1)
      {
            return new DefaultMessageStore();
      }

      public void dispose()
      {
        tf = null;
        ee = null;
        ss = null;
        vf = null;
        bf = null;

      }
}
```

Figure 8. Required Methods of IModule Interface

All value classes in Bogor must implement the IValue interface. Since stack is a non-primitive data type, it has to implement a descendent of IValue called INonPrimitiveValue interface. The required methods of INonPrimitiveValue interface are as shown in Figure 9 below.

```
public interface INonPrimitiveExtValue extends
                 INonPrimitiveValue, Serializable
{
      // Methods required directly

      Field[] getFields();

      byte[][] linearize(int bitsPerNonPrimitiveValue,
                         ObjectIntTable<INonPrimitiveValue>
                         nonPrimitiveValueIdMap,
                         int bitsPerThreadId,
                         IntIntTable threadOrderMap);

      void visit(IValueComparator vc,
                 boolean depthFirst,
                 Set<IValue> seen,
                 LinkedList<IValue> workList,
                 IValueVisitorAction vva);

      // Methods required by INonPrimitiveValue

      int getReferenceId();

      // Methods required by IValue

      Type getType();

      int getTypeId();

      INonPrimitiveExtValue clone(Map<Object,Object>
                                              cloneMap);

      void validate(IBogorConfiguration bc);

      public boolean equals(Object o);

      public int compareTo(IValue o);
}
```

Figure 9. Required Methods of INonPrimitiveValue Interface

The linearize ( ) method encodes a state into a bit sequence that uniquely represents a state in the state space. The Visit ( ) and getFields ( ) methods are used by Bogor for several analytical purposes. The complete stack implementation is presented in Appendix C.

4.1.3 Implementation and Experimental Results

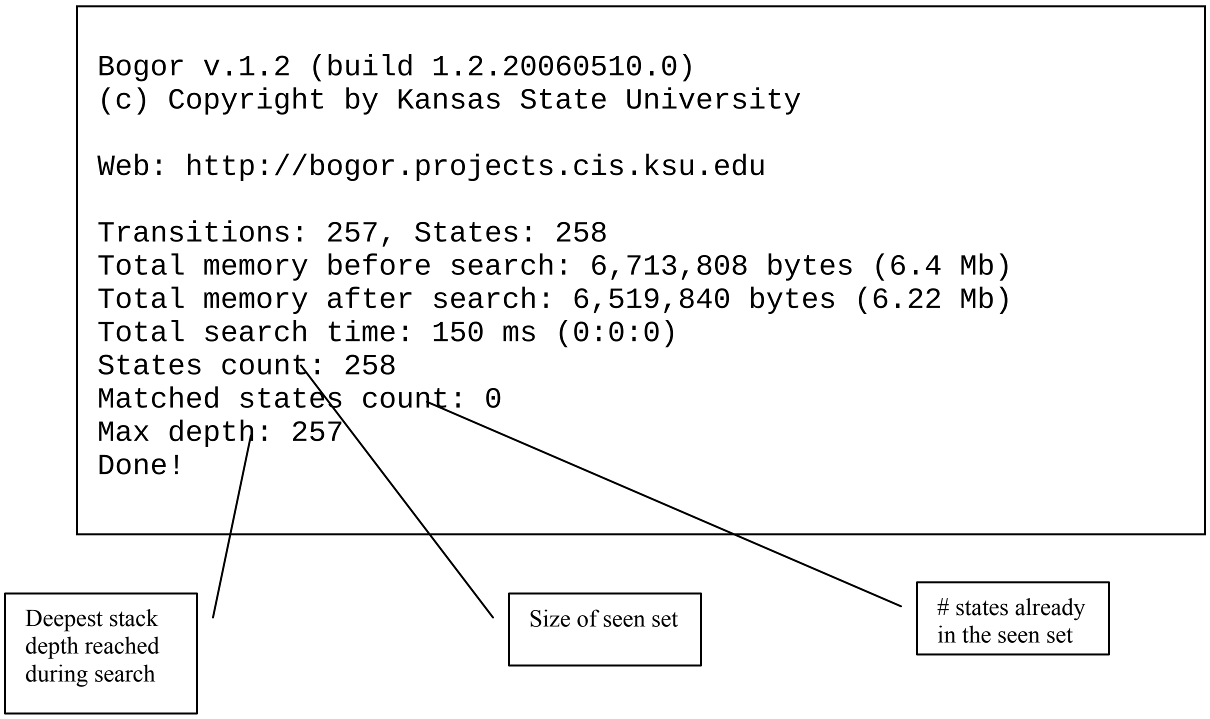
The first step in model checking is to represent a system as a model. The Postfix Calculator system was represented using Bogor's input language BIR. The model uses the ADT Stack extension that was developed as a part of this thesis. A BIR excerpt of the implementation of the Postfix Calculator appears below in Figure 10.

```
extension Stack for bogor.MyStack.myStackModule
{
      . . .
}

Stack.type<int> operands;

main thread MAIN()
{
      // READ THE POSTFIX EXPRESSION
      GetExpression();


      // CREATE STACK USING EXTENSIONS
      operands := Stack.create<int>(1);


      // EVALUATE POSTFIX EXPRESSION
      Evaluate();


function Evaluate() returns int
{
      . . .

      if (c == Operand)
      do
      // Pop 2 elements, operate, and push the result
            op2 := Stack.pop<int>(operands);
            op1 := Stack.pop<int>(operands);
            Stack.push<int>(operands, result);

      else do
      // PUSH THE ELEMENT
            Stack.push<int>(operands, c);
}
```

Figure 10. BIR Excerpt for the Postfix Calculator Model

To ensure the proper functioning of the system, a simple postfix expression was fed as an input to the model. The model was checked for the accuracy of the result, working of basic operations (push, pop, etc., for stack extension), and the working of the extension mechanism.

The trial runs were conducted on Windows XP and Solaris 9.0 operating systems. Bogor was run on a Java 2 Platform and the results were observed on Eclipse 3.2.0. The Postfix Calculator model ran successfully and the result of the model checking is given in Figure 11 below.



```
Bogor v.1.2 (build 1.2.20060510.0)
(c) Copyright by Kansas State University

Web: http://bogor.projects.cis.ksu.edu

Transitions: 257, States: 258
Total memory before search: 6,713,808 bytes (6.4 Mb)
Total memory after search: 6,519,840 bytes (6.22 Mb)
Total search time: 150 ms (0:0:0)
States count: 258
Matched states count: 0
Max depth: 257
Done!
```

Deepest stack depth reached during search
Size of seen set
# states already in the seen set

(Seen set stores states that are explored in a computation tree, to avoid revisiting them.)

Figure 11. Model Checking Result of the Postfix Calculator

The next step in Model Checking is to add the specifications that the system needs to check. At no point of time one would want to pop up elements from an empty stack.

Also, when an operand is encountered, at least two elements are to be present in the stack. Assertions are inserted at proper points in the model to check for these properties. An assertion is a boolean that checks to determine if a condition holds true, and it flags an error if the condition is not satisfied.

```
if (c == Operand) do

//CHECKS IF STACK HAS AT LEAST TWO ELEMENTS
      assert (Stack.size<int>(resources)>= 2);


//POP 2 ELEMENTS, OPERATE, AND PUSH THE RESULT


//ASSERT TO AVOID POPPING FROM AN EMPTY STACK
      assert (!Stack.isEmpty<int>(resources));
      op2 := Stack.pop<int>(operands);

//ASSERT TO AVOID POPPING FROM AN EMPTY STACK
      assert (!Stack.isEmpty<int>(resources));
      op1 := Stack.pop<int>(operands);
      Stack.push<int>(operands, result);

else do . . .
```

Figure 12. The Postfix Calculator Model with Assertions

The model was again tested to check if it meets the specifications. There were no assertion violations and this verifies the proper functioning of the model using the extensions.

Another model of the Postfix Calculator was developed in BIR using an array (a non primitive data type in BIR) as a stack, unlike the first model that used the ‘Stack’ ADT extension. A comparison was made in terms of memory usage and the time to search the state space. The complexity of the system is gradually increased by feeding it more complex postfix expressions. Table I describes the experimental results of two models showing number of transitions, number of states, memory used (in megabytes), and time (in milliseconds). Figure 13 shows the comparison of state space sizes for model using arrays as stack versus model using stack extension.

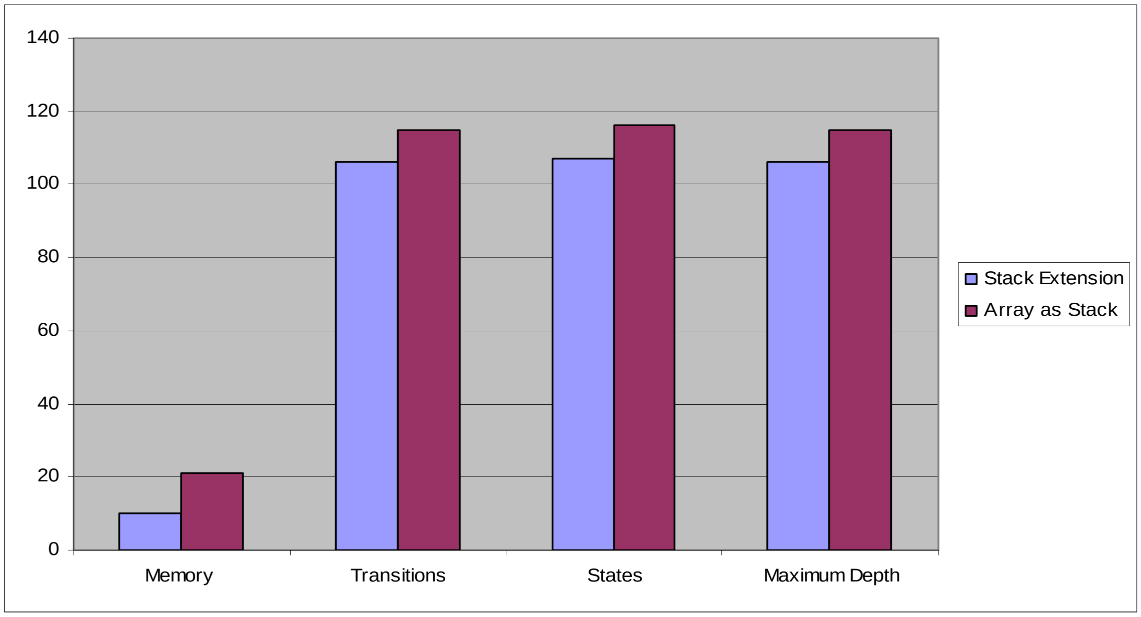
TABLE I. Experimental Results

	Postfix Expression	Time (ms)	Memory (Mb)	No. of Transitions	No. of States	Max. Depth
Exp A	**Stack Extension**	100	10.07	106	107	106
	Array as Stack	20	21.12	115	116	115
Exp B	**Stack Extension**	711	11.75	394	395	394
	Array as Stack	1312	21.85	411	412	411
Exp C	**Stack Extension**	3375	11.63	751	752	751
	Array as Stack	7771	20.69	778	779	778

Exp A = 2 3 +

Exp B = 6 3 / 4 3 * + 2 + 8 -

Exp C = 6 3 / 4 3 * + 20 + 8 9 * 3 / - 6 9 * + 12 –



(The highlighted values from TABLE I are used to plot this graph.)

Figure 13. Comparison of Resource Requirements

4.1.4 Observations

The experimental results show that there is an evident overhead in terms of memory usage when an array is used to represent a stack. However, there is no overhead in terms of time when functions are called, probably because the functions are local to the model. Thus, using an array to simulate a stack is possible but it would inflate the state space and would cause runtime penalty in terms of memory requirements, especially in larger systems. By introducing a new, encapsulated BIR native data type stack, the code complexity can be pushed into the model checker's runtime environment, thus reducing the state space.

4.2 Case Study - Resource Allocation/Deallocation (Multiset Extension)

Resource allocation is the process of allocating a resource from the pool of available resources. Once the resource is used by a process, it is put back in the pool for other processes to use.

A straightforward representation of a resource pool is a set. Set is not a native BIR construct, though it can implemented using BIR arrays. If arrays were to be used as sets, care has to be taken to ensure element uniqueness. Sets are unordered and the basic operations on them are: membership, add an element, remove an element, isEmpty, etc.

Bogor describes [Bogor 2003] an implementation of resource pool using a set. However, a better representation of a resource pool would be a multiset. A multiset differs from a set in that each member has a multiplicity, which determines how many times an element occurs in the multiset, e.g., multiset (1, 4, 4) has element 1 with multiplicity 1, represented as (1 , 1), and element 4 with multiplicity 2, represented as (4 , 2). A resource pool represented as a multiset can have more than one instance of a resource, unlike a set representation where duplicate membership is not permitted.

One of the goals of this case study was to study multiple threads that have concurrent executions. A multiset extension was developed as a part of this thesis work and the methodology for the same is described below.

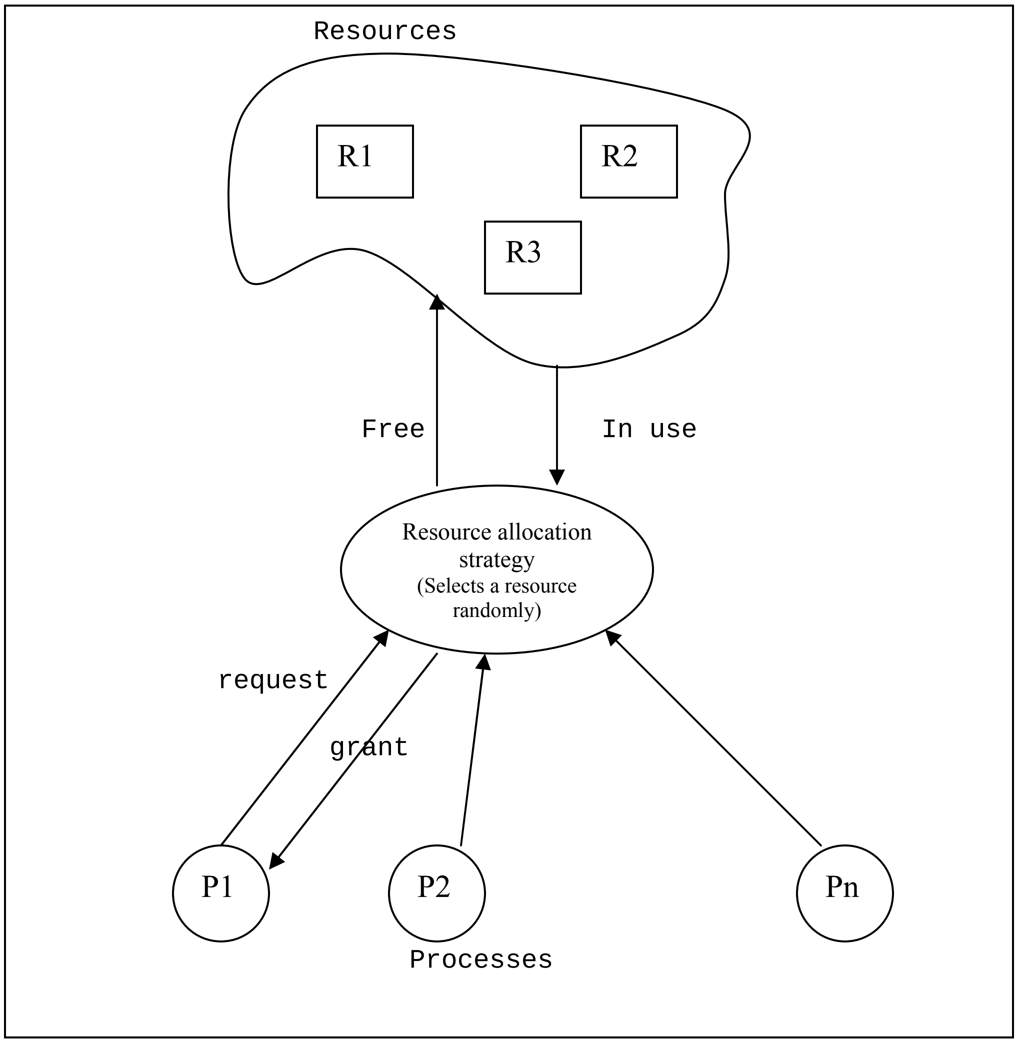


Figure 14. Resource Allocation Process

4.2.1 Extension Syntax

In order to implement the multiset extension, the following operations on multiset are required to be defined.

List of Operations:

- Create ( ) - Create a multiset with the given values
- isMember() – Determine if the given element is in the multiset
- Add ( ) - Add an element to the multiset
- Remove ( ) - Remove an element from the multiset
- isEmpty ( ) - Return a boolean if the set is empty
- frequency ( ) - Return the frequency or multiplicity of the given element

The following code segment informs the Bogor language recognizer about the new extension type.

```
extension Multiset for bogor.multiset.MultisetModule
{
      typedef type <'a>;
      expdef Multiset.type <'a> create <'a> ('a ...);
      expdef boolean isEmpty <'a> (Multiset.type <'a>);
      expdef 'a selectElement<'a>(Multiset.type<'a>);
      expdef int frequency<'a>(Multiset.type<'a>, 'a);
      actiondef add<'a>(Multiset.type<'a>, 'a);
      actiondef remove<'a>(Multiset.type<'a>, 'a);
}
```

Figure 15. Extension Definition for Multiset

The keyword ‘extension’ is followed by extension name ‘`Multiset`’, followed by java class ‘MultisetModule’. The type of this extension is kept generic and can be instantiated to any data type at the time of declaration, e.g., Multiset.type<int> MyMultiSet declares a Multiset of integers.

4.2.2 Implementation Semantics

The implementation semantics of multiset is similar to what was shown in the case study Section 4.1. The complete implementation of the multiset extension is presented in Appendix D.

4.2.3 Implementation and Experimental Results

The resource contention system was built as a BIR model to make it comprehendible to Bogor. This system uses the multiset extension described above to implement a collection of resources (a resource pool). Figure 16 shows the BIR excerpt for the same.

The model was tested against a given scenario where there are two Disks and one Display in the resource pool. There are two processes that are executed simultaneously and they have access to a common resource pool. These processes request a resource, acquire it (only if it is available), use it, and then put it back into the resource pool. The model was checked for the accuracy of the result, the working of the basic operations (add, remove, etc., for the multiset extension), and the working of extension mechanism.

```
extension Multiset for bogor.multiset.MultisetModule
{
      . . .
}

enum ResourceState { FREE, IN_USE }
record Resource { ResourceState state; }
Multiset.type<Resource> resources;
      . . .

main thread MAIN()
{
      . . .
resources :=Multiset.create<Resource>(DISK, DISPLAY, DISK);
start Process();
start Process();
}

Thread Process
{
      Invoke ManageResources();
}

function ManageResources()
{
      Resource resource;

      !Multiset.isEmpty<Resource>(resources)
      do{
      //SELECT A RESOURCE RANDOMLY
      resource :=Multiset.selectElement<Resource>(resources);

      //REMOVE THE RESOURCE FROM THE POOL OF RESOURCES
      Multiset.remove<Resource>(resources, resource);}

      //USE THE RESOURCE

      //FREE THE RESOURCE AND ADD IT BACK TO THE POOL
      Multiset.add<Resource>(resources, resource);
}
```

Figure 16. BIR Excerpt for the Resource Contention Model

The trial runs were conducted on Windows XP and Solaris 9.0 operating systems. Bogor was run on the Java 2 Platform and the results were observed on Eclipse SDK

3.2.0. The Resource Contention model ran successfully and the result of the model checking is given in Figure 17 below.

```
Bogor v.1.2 (build 1.2.20060510.0)
(c) Copyright by Kansas State University

Web: http://bogor.projects.cis.ksu.edu


Total memory before search: 7,481,880 bytes (7.14 Mb)
Total memory after search: 7,582,304 bytes (7.23 Mb)
Total search time: 170 ms (0:0:0)
States count: 119
Matched states count: 137
Max depth: 69
Done!
```

Figure 17. Model Checking Result of Resource Contention

The complexity of the resource contention model was gradually increased to test it for scalability. Table II shows the results that were obtained by varying the number of concurrent threads and the available resources.

TABLE II. Experimental Data for Resource Contention

No. of Processes	No. of Resources	Time (ms)	Memory (Mb)	No. of Transitions	No. of States	Matched States	Max. Depth
2	**2**	210	9.35	183	87	97	59
3	**2**	421	9.54	1465	487	979	248
3	**3**	711	9.57	3196	1005	2192	487
4	**2**	1312	9.80	8921	2321	6601	817
4	**3**	3375	10.50	27029	6555	20476	2474
4	**4**	7771	11.81	64141	14964	48178	5694

No. of Processes	No. of Resources	Time (ms)	Memory (Mb)	No. of Transitions	No. of States	Matched States	Max. Depth
5	**2**	6159	11.04	46541	10037	36505	2257
5	**3**	23745	15.51	180236	36014	144223	9327
5	**4**	76049	28.33	568581	108002	460580	33748
5	**5**	181120	63.14	97629	50147	47833	45523

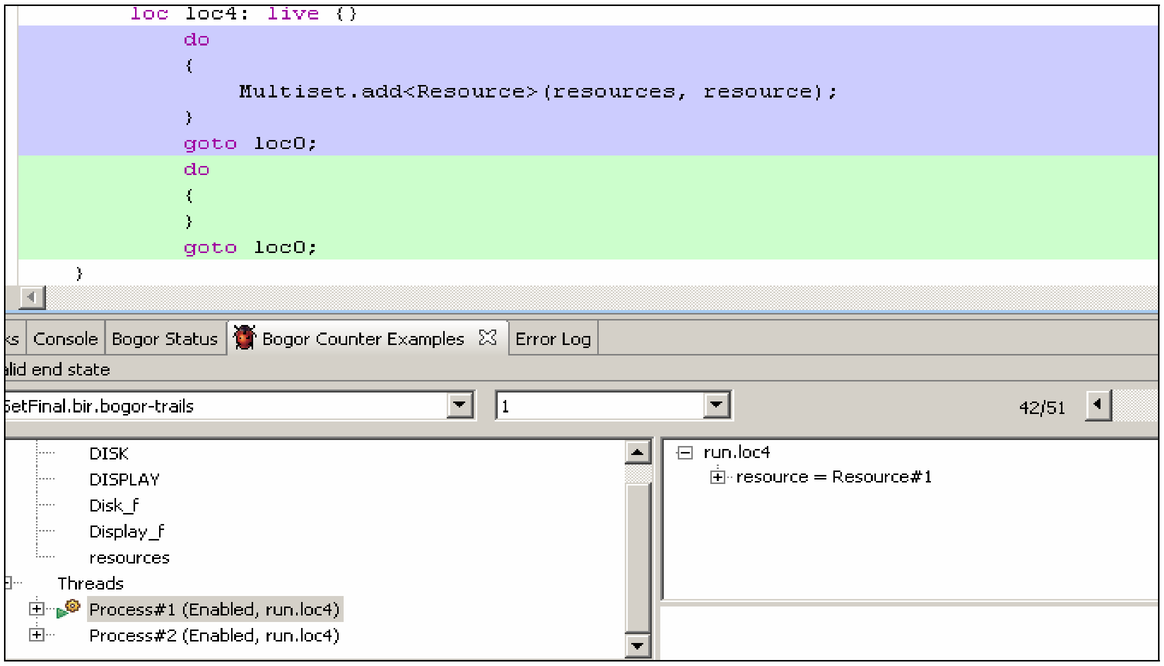
A deadlock may occur if there is a leakage in the resource pool, wherein a process after using a resource fails to put it back in the resource pool. Eventually, all the resources in the resource pool would deplete, and all the processes will end up waiting for the other processes to free the resource. Such a condition where two or more processes are waiting for resources, which are not going to be made available, is a circular chain that will result in deadlock. The above behavior was replicated in the BIR model by adding a code segment as shown in Figure 18, where a thread chooses an execution path provided in the code. By choosing path A, a thread returns the resource to the resource pool and path B causes leakage of resources.

```
//ADD RESOURCE TO THE RESOURCE POOL
{

      //PATH A
      do
      {
            Multiset.add<Resource>(resources, resource);
      }

      //OR PATH B

      do
      {
            // BUG: leak resource by not replacing it
      }
}
```

Figure 18. Resource Contention Model with Deadlock Scenario

The model was checked to test how it reacted to the possibility of deadlocks. Bogor was able to point out the deadlock caused by resource leakage. A counter-example was generated in the form of a trail file. The Eclipse platform was used to view the trail file and do a step-by-step analysis of the counter-example (see Figure 19).

To ensure the proper functioning of the system, a few other assertions were added. Care was taken to avoid a condition where the same resource is randomly selected by more than one process. A code fragment was added to the existing code that ensures the process of selecting and removing a resource from the resource pool is atomic (as given in Figure 20). An assertion was added to determine if a resource exists before it is removed. The model was again tested to check if it meets the specifications. There were no assertion violations and this verifies the proper functioning of the resource contention model using the multiset extension.



```
        loc loc4: live {}
            do
            {
                Multiset.add<Resource>(resources, resource);
            }
            goto loc0;
            do
            {
            }
            goto loc0;
    }
```

```
    DISK
    DISPLAY
    Disk_f
    Display_f
    resources
Threads
    Process#1 (Enabled, run.loc4)
    Process#2 (Enabled, run.loc4)

run.loc4
    resource = Resource#1
```

Figure 19. Screen Shot of Bogor Counter-Example in Eclipse

```
when !Multiset.isEmpty<Resource>(resources) do
{
      // SELECTION AND REMOVAL OF A RESOURCE IS MADE ATOMIC

      atomic //start of atomic statement

      resource :=
Multiset.selectElement<Resource>(resources);

      //CHECK IF A RESOURCE EXISTS BEFORE REMOVING IT
      assert
(Multiset.isMember<Resource>(resources,resource);

      Multiset.remove<Resource>(resources, resource);

      End // End of atomic statement
}
```

Figure 20. Resource Contention Model with Assertions

CHAPTER V

SUMMARY AND FUTURE WORK

5.1 Summary

The crux of this thesis was to study and experiment with domain specific approaches to software model checking.

Chapter I introduced the importance of software model checking and the scope of the thesis. Chapter II reviewed the literature of model checking and the basic problem of state space explosion. It Chapter II also described the general trends in model checking and commented briefly on the drawbacks of most of the existing model checkers.

In Chapter III, Bogor was introduced as a novel model checker with an extensible framework. The focus of Chapter III then shifted towards customization of Bogor with the help of Eclipse's PDE.

Chapter IV detailed the implementation of the extensions that are part of this thesis work. The first part of the chapter discussed the need for of an extension and its potential use in a number of applications. This chapter described two case studies using extensions stack and multiset. The later part of the chapter presented the test results and the trail files for debugging for both of the cases. The complexity of the applications was gradually increased to test for scalability and applicability.

Customizing the model checker as per domain specifications helped in achieving space reduction. The space reduction appears to be significant, especially in large domains where it contributes towards the solution of the state space explosion problem. Although development of extensions is achievable, it requires a working knowledge of Eclipse and specific knowledge of model checking.

In conclusion, a domain specific approach for software model checking has been demonstrated to be a promising methodology. Language extensions to BIR were successfully built and tested for accuracy and scalability.

5.2 Future Work

The extensions developed and reported in this thesis are fundamental with the intent of handling smaller application, but there is a lot of potential for integrating these extensions to check larger systems. However, with the successful implementation of extensions, it is evident that there is a large scope of research in the area of domain specific approach to model checking. A possible future area of work in this field would be building an entire library of basic extensions that would be supplementary to the model checker.

REFERENCES

[Biere et al. 1999] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu, "Symbolic Model Checking without BDDs", *Lecture Notes in Computer Science*, No. 1579, pp. 193-207, Springer Verlag, 1999.

[Bogor 2003] Bogor: Software model checking framework, http://bogor.projects.cis.ksu.edu, date created: 2003, date accessed: May 2005.

[Brat et al. 2000] G. Brat, K. Havelund, S. Park, and W. Visser, "Java PathFinder – A Second Generation of a Java Model Checker", *Proceedings of the Workshop on Advances in Verification*, pp. 130-135, Chicago, Illinois, July 2000.

[Burch et al. 1994] J. R. Burch, E. M. Clarke, D. E. Long, K. L. MacMillan, and D. L. Dill, "Symbolic Model Checking for Sequential Circuit Verification", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 13, No. 4, pp. 401-424, April 1994.

[Chan et al. 2001] W. Chan, R. J. Anderson, P. Beame, D. H. Jones, D. Notkin, and W. E. Warner, "Optimizing Symbolic Model Checking for Statecharts", *IEEE Transactions on Software Engineering*, Vol. 27, No. 2, pp. 170-190, February 2001.

[Clarke et al. 1999] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled, *Model Checking*, MIT Press, Cambridge, Massachusetts, 1999.

[Clarke et al. 1986] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications", *ACM Transactions on Programming Languages and System (TOPLAS)*, Vol. 8, No. 2, pp. 244-263, April 1986.

[Clarke et al. 1989] E. M. Clarke, D. Long, and K. McMillan, "Compositional Model Checking", *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pp. 353-362, Pacific Grove, California, June 1989.

[Clarke et al. 1994] E. M. Clarke, O. Grumberg, and D. E. Long, "Model Checking and Abstraction", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 16, No. 5, pp. 1512-1542, September 1994.

[Clarke and Mishra 1983] E. Clarke and B. Mishra, "Automatic Verification of Asynchronous Circuits", *Proceedings of Logic of Programs*, pp. 101-115, Pittsburgh, PA, June 1983.

[DeLoach and Matson 2004] S. A. DeLoach, and E. Matson, "An Organizational Model for Designing Adaptive Multiagent Systems", *Proceedings of the AAAI-04 Workshop on Agent Organizations: Theory and Practice*, San Jose, California, July 2004.

[Demartini et al. 1999] C. Demartini, R. Iosif, and R. Sisto, "dSPIN: A Dynamic Extension of SPIN" *Proceedings of the 6th International SPIN Workshop*, Vol. 1680 of Lecture Notes in Computer Science, pp. 261–276, Springer-Verlag, September 1999.

[Dwyer et al. 2005] Matthew B. Dwyer, John Hatcliff, Matthew Hoosier, and Robby, "Building Your Own Software Model Checker Using the Bogor Extensible Model Checking Framework", *Proceedings of the 17th Conference on Computer-Aided Verification (CAV 2005),* Edinburgh, Scotland, UK, July 2005.

[Eclipse 2004] Eclipse: Platform for tools integration, http://www.eclipse.org/, date created: 2004, date accessed: October 2005.

[Emerson 1990] E. A. Emerson, "Temporal and Modal Logic", *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, Elsevier Science Publishers B.V., pp. 997-1072, 1990.

[Eleftherakis and Kefalas 2001] G. Eleftherakis and P. Kefalas, "Towards Model Checking of Finite State Machines Extended with Memory through Refinement", *Advances in Signal Processing and Computer Technologies*, pp. 321-326. World Scientific and Engineering Society Press, July 2001.

[Edelkamp et al. 2001] S. Edelkamp, A. L. Lafuente, and S. Leue, "Directed Explicit Model Checking with hsf-spin", *Proceedings of the 8th International SPIN Workshop,* Vol. 2057 of Lecture Notes in Computer Science, pp. 57-79, Toronto, Canada, May 2001.

[Godefroid 1997] P. Godefroid, "Model Checking for Programming Languages Using Verisoft", *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, *(POPL'97)*, pp. 174-186, Paris, France, January 1997.

[Godefroid and Pirottin 1993] P. Godefroid and D. Pirottin, "Refining Dependencies Improves Partial-Order Verification Methods", *Proceedings of the 5th Conference on Computer-Aided Verification*, Lecture Notes in Computer Science 697, pp. 438-449, Elounda, Greece, June 1993.

[Groce and Visser 2002] A. Groce and W. Visser, "Model Checking Java Programs Using Structural Heuristics", *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 12-21, Rome, Italy, July 2002.

[Holzmann 1997] G. J. Holzmann, "The Model Checker SPIN", *IEEE Transactions on Software Engineering*, Vol. 23, No. 5, pp. 279-294, Boston, Massachusetts, May 1997.

[Hoosier et al. 2004] Matthew Hoosier, John Hatcliff, Robby, and Matthew B. Dwyer, "A Case Study in Domain Customized Model Checking for Real-Time Component Software", *Proceedings of the 1st International Symposium on Leveraging Applications of Formal Method (ISoLA),* Paphos, Cyprus , November 2004.

[Manna and Pnueli 1992] Zohar Manna and Amir Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification*, Springer-Verlag, New York, NY, 1992.

[McMillan 1992] K. L. McMillan, "Symbolic Model Checking: An Approach to the State Explosion Problem", Ph.D. Thesis, Computer Science Department, Carnegie Mellon University, CMU-CS-92-131, 1992.

[Queille and Sifakis 1982] Jean Pierre Queille and Joseph Sifakis, "Specification and Verification of Concurrent Systems in CESAR", *Proceedings of the Fifth International Symposium in Programming*, Vol. 137 of Lecture Notes in Computer Science, pp. 337-351, New York, NY, April 1982.

[Robby et al. 2003a] Robby, Matthew B. Dwyer, J. Hatcliff, and R. Iosif, "Space Reduction Strategies for Model Checking Dynamic Systems", *Proceedings of the 2003 Workshop on Software Model Checking*, Boulder, Colorado, July 2003.

[Robby et al. 2003b] Robby, Matthew B. Dwyer, and John Hatcliff, "Bogor: An Extensible and Highly Modular Software Model Checking Framework", *Proceedings of the 9th European Software Engineering Conference* held jointly with the *11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 267-276, Helsinki, Finland, September 2003.

[Robby et al. 2004] Robby, E. Rodrguez, M. Dwyer, and J. Hatcliff, "Checking Strong Specifications Using an Extensible Software Model Checking Framework", *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, pp. 404-420, Barcelona, Spain, January 2004.

[Robby et al. 2006] Robby, Scott A. DeLoach, and Valeriy A. Kolesnikov, "Using Design Metrics for Predicting System Flexibility", *Proceedings of the 2006 International Conference on Fundamental Approaches to Software Engineering (FASE)*, Vienna, Austria, March 2006.

[Schuppan and Biere 2004] V. Schuppan, and A. Biere, “Efficient Reduction of Finite State Model Checking to Reachability Analysis”, *International Journal on Software Tools on Technology Transfer (STTT)*, pp. 185-204, May 2004.

[Visser et al. 2000] W. Visser, K. Havelund, G. Brat, and S. Park, “Model Checking Programs”, *Proceedings of the International Conference on Automated Software Engineering*, pp. 3-12, Grenoble, France, September 2000.

APPENDICES

APPENDIX A

GLOSSARY

ADT	Abstract Data Type is a specification of a set of data and the set of operations that can be performed on the data.
API	Application Program Interface is a set of routines, protocols, and tools for building applications.
BIR	Bandera Intermediate Representation is a language used by the Bogor model checker to represent models of concurrent object-oriented systems.
Computation Tree	A tree is formed by unwinding the Kripke Structure from the initial state. The computation tree shows all possible executions starting from the initial state.
Concurrent Program	A program that is made up of several processes/task/threads whose execution can be multiplexed and/or done in parallel.
CORBA	Common Object Request Broker Architecture is a standard defined by the Object Management Group (OMG) that enables software components written in multiple computer languages and running on multiple computers to interoperate.
CTL*	Computation Tree Logic describes properties of computation trees. The sublogics of CTL* are branching time logic and linear time logic.
Eclipse	An extensible development platform for building software.
Extensible Framework	A framework that provides facilities for adding new features.

GUI	Graphical User Interface refers to a front-end that provides an attractive and easy-to-use interface for interacting with an application.
IDE	Integrated Development Environment is a type of computer software that assists computer programmers in developing software.
JRE/JDK/JVM	Java Runtime Environment (JRE) is a subset of Java Development Kit (JDK) that contains the core executables and files that constitute the standard Java platform. JRE includes Java Virtual Machine (JVM), core classes, and supporting files.
Model Checking	An automatic technique for verifying finite state concurrent systems.
Path Quantifier	Path Quantifiers 'A' (for all paths) and 'E' (for some paths) are used to describe the branching structure in a computation tree.
PDE	Plug-in Development Environment provides tools to create, develop, test, debug, build, and deploy Eclipse plug-ins, fragments, and features.
Reactive System	A system that changes its actions, outputs, and states in response to stimuli from within or from outside. It is an event driven or control driven system continuously having to react to external and/or internal stimuli.
Temporal Logic	A formalism used to describe a system in terms of propositions and temporal qualifiers.
Transition Diagram	A graphical structure indicating possible states of a system and transitions from one state to another.

APPENDIX B

LIST OF SOME OF THE POPULAR MODEL CHECKERS AND THEIR FEATURES, INPUT LANGUAGE, GRAPHICAL INTERFACE, AND PLATFORMS

Model Checker	Features	Input Language	Graphical Interface	Platform
SPIN	System is seen as a synchronized Extended Finite State Machine (EFSM). Used for modeling for communication protocol and LTL model.	Promela	**Yes**	Unix, Solaris, and Linux machines, on most flavors of Windows PCs, and on Macs
NuSMV	BDD based CTL and LTL model checkers (under fairness), bounded model checking with LTL. Used for verifying digital circuits	Symbolic Model Verifier (SMV) input language, a simple circuit description language	**Yes**	MS Windows XP Linux RedHat 9.0.
Verisoft	A tool for software developers and testers of concurrent/reactive/real-time systems	C, C++, Tcl	**Yes**	Solaris/Sparc and Linux
MAGIC	Checks conformance between component specifications and their implementations	C	**No**	RedHat 7.1, RedHat 8.0 and Windo ws 2K

Java Path Finder	Verifies executable Java bytecode programs.	Java	**No**	Windows, Unix related
CBMC	Bounded Model Checker for C (CBMC) is used for embedded software, and supports dynamic memory allocation	ANSI-C, SMV, Verilog, and netlists	**Yes**	Windows, Unix related
BLAST	(Berkeley Lazy Abstraction Software Verification Tool) BLAST model checks C programs and uses automatic abstraction to construct models.	C	**No**	Windows, Unix related
MOCHA	MOCHA is an interactive software environment for system specification and verification that exploits design structure in automatic verification.	Reactive modules	**Yes**	Windows, Unix related
UPPAAL	It stands for UPP (Uppsala University) + AAL (Aalborg University). Uppaal is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata.	Automata	**Yes**	Windows, Unix related
CADP	Construction and Analysis of Distributed Processes (CADP) is a toolbox for the design of communication protocols and distributed systems	C	**Yes**	Windows, Unix related
Bandera	Bandera tool set is used for model checking concurrent Java software	Java	**Yes**	Windows, Unix related
Cadence SMV	SMV can be used as a learning tool to introduce the general principles of model checking and refinement verification.	Extended SMV, Verilog	**Yes**	Windows, Unix related

Bogor	Bogor is an extensible software model checking framework with state of the art software model checking algorithms, visualizations, and user interface	BIR	**Yes**	Unix related

APPENDIX C

PROGRAMS FOR CASE STUDY - POSTFIX CALCULATOR

(STACK EXTENSION)

This appendix contains the following six code listings showing the BIR representation of the Postfix Calculator system using stack extension, using array as stack, and the java classes that implement the stack extension.

a) Algorithm for the Postfix Calculator
b) Postfix Calculator using stack extension
c) Postfix Calculator using array as stack
d) MyStackModule.java
e) MyIStackValue.java
f) MyDefaultStackValue.java

a) Algorithm for the Postfix Calculator:

```
// Exp is a postfix expression of length n
// Start
//    for i = 1 to n begin
//          if isoperand( Exp(i) )
                  then push(value of Exp(i) )
//          else
//                if isoperator( Exp(i) ) then begin
//                      op2 = pop
//                      op1 = pop
//                      and push result (note the order)
//                end
//          end
//    x = pop
//    return x
// end.
```

b) Postfix Calculator using stack extension

```
      system TestStackArray
      {
            }



            extension Stack for bogor.MyStack.myStackModule
            {
                  typedef type<'a>;
                  expdef boolean isEmpty<'a>(Stack.type<'a>);
                  expdef Stack.type <'a> create <'a> ('a);
                  expdef int size<'a>(Stack.type<'a>);
                  expdef 'a getTop<'a>(Stack.type<'a>);
                  actiondef push<'a>(Stack.type<'a>, 'a);
                  actiondef pop<'a>(Stack.type<'a>);
            }



            Stack.type<int> resources;
            boolean initialized;
            int t;

            int[] Exp;
            int length;
            int c;
            int i;
            int result;

            function GetExpression ()
            {
                  loc loc0:
                  do
                  {
                        Exp := new int[50];
                        length := 3;
                        Exp[0]:= 2; Exp[1]:=3; Exp[2]:='+'; */
                  }
                  return;
            }


            main thread MAIN()
            {
                  //read the Postfix Expression
                  loc loc0: live {}
                              invoke GetExpression()
                  goto loc1;

                  //create Stack using extensions
                  loc loc1: live{}
                  do
                  {
                  resources := Stack.create<int>(-1);
                  }
                  goto loc2;

                  //evaluate Postfix expression
                  loc loc2: live{}
                  invoke Evaluate()
                  return;
            }
```

```
        function Evaluate() returns int
        {
              int op1;
              int op2;

              i := 0;
              while i < length do
                    c:= Exp[i];
                    i := i + 1;
                    if (c == '*'||c == '+'||c == '/'||c == '-') do
                    //check if stack is empty
                    assert (!Stack.isEmpty<int>(resources));

                    //pop two elements, operate and push the result
                    op2 := Stack.getTop<int>(resources);
                    Stack.pop<int>(resources);
                    op1 := Stack.getTop<int>(resources);
                    Stack.pop<int>(resources);

                    if (c == '*') do
                          result := op1 * op2;
                    end
                    if (c == '+') do
                          result := op1 + op2;
                    end
                    if (c == '/') do
                          result := op1 / op2;
                    end
                    if (c == '-') do
                          result := op1 - op2;
                    end
                    Stack.push<int>(resources, result);
                    else do
                    //push the element
                    Stack.push<int>(resources, c);

                    end   //end of ‘if’ statement
              end   // end of ‘while do’ statement

              return result;

        }
  }
```

c) Postfix Calculator using array as stack

```
  system PostfixCalc
  {
        int[] Exp;
        int length;
        int[] Stack;
        int Top;
        int c;
        int result;
        int i;
```

```
/*This function creates an expression */
function CreateExp ()
{
      loc loc0:
      do
      {
            Exp := new int[50];
            length := 3;
            Exp[0]:= 2; Exp[1]:=3; Exp[2]:='+';
      }
      return;
}

/*This function creates a stack of the given size */
function Create (int size)
{
      loc loc0:
      do
      {
            Stack := new int[size];
            Top:= 0;
      }
      return;
}

/*This function pushes a given element into the stack */
function Push (int element)
{
      loc loc0:
      do
      {
            Stack[Top] := element;
            Top := Top + 1;
      }
      return;
}

/*This function pops the top elemnt of the stack*/
function Pop () returns int
{
      int TopElement;
      loc loc0:
      do
      {
            Top:= Top -1;
            TopElement:= Stack[Top];

      }
      return TopElement;
}

/*This function returns the top of the stack */
function GetTop() returns int
{
      int TopElement;
      loc loc0:
      do
      {
            TopElement:= Stack[Top];
      }
      return TopElement;
}

/*This function returns true value if the stack is empty*/
function IsEmpty() returns boolean
{
      boolean empty;
      loc loc0:
      do
      {
            empty := Top == 0;
      }
      return empty;
}

/*This function returns the size of the stack*/
function Size() returns int
{
      int size;
      loc loc0:
      do
      {
            size := Top + 1;
      }
      return size;
}

active thread MAIN()
{
      CreateExp();
      Create(50);
      Evaluate();
      return;
}

/*This function evaluates the postfix expression*/
function Evaluate() returns int
{
      i := 0; j:= 0;
      while i < length do
            c:= Exp[i];
            i := i + 1;
            if (c == '*'||c == '+'|| c == '/'||c == '-') do
            OP();// := invoke Push() return;
            else do
            Push(c);
            end
      end

      return result;
}

/*This function is invoked if an operand is encountered*/
function OP()
```

```
        {
              int op1;
              int op2;
              op2 := Pop();
              op1 := Pop();
              if (c == '*') do
                    result := op1 * op2;
              end
              if (c == '+') do
                    result := op1 + op2;
              end
              if (c == '/') do
                    result := op1 / op2;
              end
              if (c == '-') do
                    result := op1 - op2;
              end
              Push(result);
              return;
        }

  }
```

d) myStackModule.java

```
package bogor.MyStack;

import java.util.Map;
import java.util.Properties;
import edu.ksu.cis.projects.bogor.IBogorConfiguration;
import edu.ksu.cis.projects.bogor.ast.Action;
import edu.ksu.cis.projects.bogor.ast.Node;
import edu.ksu.cis.projects.bogor.module.DefaultMessageStore;
import
edu.ksu.cis.projects.bogor.module.IBacktrackingInfoFactory;
import edu.ksu.cis.projects.bogor.module.IExpEvaluator;
import edu.ksu.cis.projects.bogor.module.IExtArguments;
import edu.ksu.cis.projects.bogor.module.IMessageStore;
import edu.ksu.cis.projects.bogor.module.IModule;
import edu.ksu.cis.projects.bogor.module.ISchedulingStrategist;
import
edu.ksu.cis.projects.bogor.module.ISchedulingStrategyContext;
import edu.ksu.cis.projects.bogor.module.ISchedulingStrategyInfo;
import edu.ksu.cis.projects.bogor.module.IValueFactory;
import
edu.ksu.cis.projects.bogor.backtrack.IActionBacktrackingInfo;
import
edu.projects.bogor.backtrack.ITransformationBacktrackingInfo;
import edu.ksu.cis.projects.bogor.module.state.IState;
import
edu.ksu.cis.projects.bogor.throwable.NullPointerBogorException;
import edu.ksu.cis.projects.bogor.module.value.IIntValue;
import edu.ksu.cis.projects.bogor.module.value.INullValue;
import edu.ksu.cis.projects.bogor.module.value.IValue;
import edu.ksu.cis.projects.bogor.type.NonPrimitiveExtType;
import edu.ksu.cis.projects.bogor.type.TypeFactory;
import edu.ksu.cis.projects.bogor.type.Type;


public class myStackModule implements IModule {


    protected TypeFactory tf;

    protected IExpEvaluator ee;

    protected IValueFactory vf;

    protected IBacktrackingInfoFactory bf;

    protected ISchedulingStrategist ss;

      public IMessageStore connect(IBogorConfiguration bc) {
            tf = bc.getSymbolTable().getTypeFactory();
            ee = bc.getExpEvaluator();
            ss = bc.getSchedulingStrategist();
            vf = bc.getValueFactory();
            bf = bc.getBacktrackingInfoFactory();
            return new DefaultMessageStore();
      }

      public String getCopyrightNotice() {
            return null;
      }

      public IMessageStore setOptions
                              (String arg0, Properties arg1) {
            return new DefaultMessageStore();
      }

      public void dispose() {
            tf = null;
            ee = null;
            ss = null;
            vf = null;
            bf = null;
      }

      public myIStackValue create(IExtArguments arg)
      {
            Type stackType = arg.getExpType();

            myIStackValue result = new
                     myDefaultStackValue(vf,(NonPrimitiveExtType)
                                 stackType, vf.newReferenceId());
            return result;
      }

      public IIntValue isEmpty(IExtArguments arg)
      {
        //gets the stack
        if (arg.getArgument(0) instanceof INullValue)
        {
```

```
        throw new NullPointerBogorException();
    }
    myIStackValue stack = (myIStackValue) arg.getArgument(0);

    //returns a boolean depending on emptiness
    return getBooleanValue(stack.isEmpty());
   }

  protected IIntValue getBooleanValue(boolean b)
  {
      return vf.newIntValue(tf.getBooleanType(), b ? 1 : 0);
  }
  public IValue getTop (IExtArguments arg)
  {
        //gets the Stack
        if (arg.getArgument(0) instanceof INullValue)
    {
        throw new NullPointerBogorException();
    }
        myIStackValue stack=myIStackValue)arg.getArgument(0);
        IValue top_element = stack.getTop();

        return (top_element);
  }

  //push operation
  public IActionBacktrackingInfo push (IExtArguments arg)
  {
        //gets the Stack
        if (arg.getArgument(0) instanceof INullValue)
        {
        throw new NullPointerBogorException();
        }
        myIStackValue stack=
        (myIStackValue)arg.getArgument(0);

        //get the element to be pushed
        IValue element = (IValue) arg.getArgument(1);
        ISchedulingStrategyContext ssc =
         arg.getSchedulingStrategyContext();
        stack.push(element);
        //create the backtracking info
        return new StackPushBacktrackingInfo(
            arg.getContainingTransition(),
            stack,
            element,
            arg.getNode(),
            ssc.getStateId(),
            ssc.getThreadId(),
            arg.getSchedulingStrategyInfo());

  }

  public IActionBacktrackingInfo pop (IExtArguments arg)
  {
        //get the Stack
        if (arg.getArgument(0) instanceof INullValue)
        {
        throw new NullPointerBogorException();
        }
        myIStackValue stack=(myIStackValue)
              arg.getArgument(0);
        ISchedulingStrategyContext ssc =
         arg.getSchedulingStrategyContext();

        if (!stack.isEmpty())
        {
              IValue element = (IValue)stack.pop();

              return new StackPopBacktrackingInfo(
                  arg.getContainingTransition(),
                  stack,
                  element,
                  arg.getNode(),
                  ssc.getStateId(),
                  ssc.getThreadId(),
                  arg.getSchedulingStrategyInfo());
        }
        else
        {
        //Create no change BacktrackingInfo
        return bf.createNoChangeBacktrackingInfo(arg
                  .getContainingTransition(), (Action)
                    arg.getNode(), arg
                  .getSchedulingStrategyInfo());
        }
  }

  public static interface IStackPushBacktrackingInfo
                  extends IActionBacktrackingInfo
  {
  }

  public static interface IStackPopBacktrackingInfo
extends IActionBacktrackingInfo
  {
  }

  protected static class StackPushBacktrackingInfo
implements IStackPushBacktrackingInfo
  {
        ITransformationBacktrackingInfo parent;

        myIStackValue stack;

        IValue element;

        Node node;

        int stateId;

        int threadId;

        ISchedulingStrategyInfo ssi;
        /**
        * Public constructor
        */
public StackPushBacktrackingInfo(
    final ITransformationBacktrackingInfo parent,
    final myIStackValue stack,
    final IValue element,
    final Node node,
    final int stateId,
    final int threadId,
    final ISchedulingStrategyInfo ssi)
  {
    this.parent = parent;
    this.stack = stack;
    this.element = element;
    this.node = node;
    this.stateId = stateId;
    this.threadId = threadId;
    this.ssi = ssi;
  }

  /**
  * Cloning constructor
  */
  private StackPushBacktrackingInfo()
  {
  }

  public ITransformationBacktrackingInfo getParent()
  {
    return parent;
  }

  public Node getNode()
  {
    return node;
  }

  public ISchedulingStrategyInfo getSchedulingStrategyInfo()
  {
    return ssi;
  }

public int getStateId()
{
    return stateId;
}

public int getThreadId()
{
    return threadId;
}

public void backtrack(IState state)
{
  stack.pop();
```

```
    }

public IActionBacktrackingInfo clone(Map<Object, Object>
                                          cloneMap)
    {
       StackPushBacktrackingInfo bi = (StackPushBacktrackingInfo)
      cloneMap
            .get(this);

        if (bi != null)
        {
            return bi;
        }

        bi = new StackPushBacktrackingInfo();
        cloneMap.put(this, bi);

        bi.element = element.clone(cloneMap);
        bi.node = node;
        bi.parent = parent.clone(cloneMap);
        bi.stack = stack.clone(cloneMap);
        bi.ssi = ssi.clone(cloneMap);
        bi.stateId = stateId;
        bi.threadId = threadId;

        return bi;
      }

      public void dispose()
      {
      }
}

      protected static class StackPopBacktrackingInfo
                        implements IStackPopBacktrackingInfo
      {
            ITransformationBacktrackingInfo parent;

            myIStackValue stack;

            Node node;

            IValue element;

            int stateId;

            int threadId;

            ISchedulingStrategyInfo ssi;

            /**
            * Public constructor
            */
      public StackPopBacktrackingInfo(
        final ITransformationBacktrackingInfo parent,
        final myIStackValue stack,
        final IValue element,
        final Node node,
        final int stateId,
        final int threadId,
        final ISchedulingStrategyInfo ssi)
      {
        this.parent = parent;
        this.stack = stack;
        this.element = element;
        this.node = node;
        this.stateId = stateId;
        this.threadId = threadId;
        this.ssi = ssi;
       }

      /**
      * Cloning constructor
      */
      private StackPopBacktrackingInfo()
      {
      }

      public ITransformationBacktrackingInfo getParent()
      {
        return parent;
      }

      public Node getNode()
      {
        return node;
      }

      public ISchedulingStrategyInfo getSchedulingStrategyInfo()
      {
        return ssi;
      }

      public int getStateId()
      {
        return stateId;
      }

      public int getThreadId()
      {
        return threadId;
      }

      public void backtrack(IState state)
      {
        stack.push(element);
      }

      public IActionBacktrackingInfo clone(Map<Object, Object>
      cloneMap)
      {
        StackPopBacktrackingInfo bi =
                  (StackPopBacktrackingInfo) cloneMap
            .get(this);
```

```
            if (bi != null)
            {
                return bi;
            }
            bi = new StackPopBacktrackingInfo();
            cloneMap.put(this, bi);
            bi.element = element.clone(cloneMap);
            bi.node = node;
            bi.parent = parent.clone(cloneMap);
            bi.stack = stack.clone(cloneMap);
            bi.ssi = ssi.clone(cloneMap);
            bi.stateId = stateId;
            bi.threadId = threadId;
            return bi;
          }

          public void dispose()
          {
          }
          }
      }
```

e) myIStackValue.java

```
      package bogor.MyStack;

      import java.util.Map;
      import edu.ksu.cis.projects.bogor.value.INonPrimitiveExtValue;
      import edu.ksu.cis.projects.bogor.module.value.IValue;

      public interface myIStackValue
      extends INonPrimitiveExtValue
      {

          //push an element
          void push(IValue v);

          //pop an element
          IValue pop();

          //determine whether this stack is empty
          boolean isEmpty();

          //get the top element of this stack
          IValue getTop();

          //determine the size of the stack
          void size();

          // specialize return type of clone
          myIStackValue clone(Map<Object, Object> cloneMap);
      }
```

f) myDefaultStackValue.java

```
package bogor.MyStack;

import java.util.Arrays;
import java.util.Comparator;
import java.util.LinkedList;
import java.util.Map;
import java.util.Set;
import java.util.Stack;
import edu.ksu.cis.projects.bogor.IBogorConfiguration;
import edu.ksu.cis.projects.bogor.module.IValueFactory;
import
edu.ksu.cis.projects.bogor.module.value.INonPrimitiveValue;
import edu.ksu.cis.projects.bogor.module.value.IValue;
import edu.ksu.cis.projects.bogor.module.value.IValueComparator;
import
edu.ksu.cis.projects.bogor.module.value.IValueVisitorAction;
import edu.ksu.cis.projects.bogor.type.NonPrimitiveExtType;
import edu.ksu.cis.projects.bogor.type.Type;
import edu.ksu.cis.projects.bogor.util.BitBuffer;
import edu.ksu.cis.projects.bogor.util.Util;
import edu.ksu.cis.projects.trove.custom.IntIntTable;
import edu.ksu.cis.projects.trove.custom.ObjectIntTable;


public class myDefaultStackValue implements myIStackValue {

      protected IValueFactory vf;

      protected NonPrimitiveExtType type;

      protected int referenceId;

      protected Stack<IValue> stack = new Stack<IValue>();

      //constructor
      public myDefaultStackValue(
              IValueFactory vf,
              NonPrimitiveExtType type,
              int referenceId)
          {
              this.vf = vf;
              this.type = type;
              this.referenceId = referenceId;
          }


      //push an element
      public void push(IValue v)
      {
            stack.push(v);
      }

       //pop an element
      public IValue pop()
      {
            return stack.pop();
      }

      //determine whether this stack is empty
      public boolean isEmpty()
      {
            return stack.empty();
      }

      //get the top element of this stack
      public IValue getTop()
      {
            return stack.peek();
      }

      public IValue[] elements()
      {
        IValue[] elements = stack.toArray(new
IValue[stack.size()]);
        orderValues(elements);

        return elements;
      }


      public myIStackValue clone(Map<Object, Object> cloneMap) {
                  myDefaultStackValue result =
      (myDefaultStackValue) cloneMap.get(this);

        if (result != null)
        {
            return result;
        }

        result = new myDefaultStackValue(vf, type, referenceId);
        cloneMap.put(this, result);

        for (IValue elem : stack)
        {
            result.push(elem);
        }

        return result;
      }

      public int getReferenceId() {
            return referenceId;
      }

      public Type getType() {
            return type;
      }

      public int getTypeId() {
            return type.getTypeId();
      }
```

```
public void validate(IBogorConfiguration bc) {
      type = (NonPrimitiveExtType) bc
  .getSymbolTable()
  .getTypeIdTypeTable()
  .get(type.getTypeId());

vf = bc.getValueFactory();
}

protected void orderValues(IValue[] values)
{
  Arrays.sort(values, new Comparator<IValue>()
      {
          public int compare(IValue o1, IValue o2)
          {
              return o1.compareTo(o2);
          }

          public boolean equals(Object obj)
          {
              return this == obj;
          }
      });
}

public void dispose() {
      if (stack != null)
        {
            stack.clear();
            stack = null;
        }

        this.vf = null;
}

public int compareTo(IValue o) {
      if (o == null)
        {
            throw new NullPointerException();
        }

        // all IValue's are "less than" other objects
        if (!(o instanceof IValue))
        {
            return -1;
        }

        // compare based on type id
        int typeComp = Util.compare
            (getTypeId(), ((IValue) o).getTypeId());

        if (typeComp != 0)
        {
            return typeComp;
        }
```

```
                myDefaultStackValue other = (myDefaultStackValue)
o;

                return Util.compare
                    (getReferenceId(), other.getReferenceId());
            }

        public byte[][] linearize(
                int bitsPerNonPrimitiveValue,
                ObjectIntTable<INonPrimitiveValue>
               nonPrimitiveValueIdMap,
                int bitsPerThreadId,
                IntIntTable threadOrderMap)
            {
                BitBuffer bb = new BitBuffer();

                IValue[] sortedElements = elements();

                vf.newVariedValueArray(sortedElements).linearize(
                    false,
                    bitsPerNonPrimitiveValue,
                    nonPrimitiveValueIdMap,
                    bitsPerThreadId,
                    threadOrderMap,
                    null,
                    bb);

                return new byte[][]
                    {
                        bb.toByteArray()
                    };
            }

        public void visit(final IValueComparator vc,
                boolean depthFirst,
                Set<IValue> seen,
                LinkedList<IValue> workList,
                IValueVisitorAction vva)
            {
                IValue[] elements = elements();

                if (depthFirst)
                {
                    for (int i = elements.length - 1; i >= 0; i--)
                    {
                        workList.addFirst(elements[i]);
                    }
                }
                else
                {
                    for (int i = 0; i < elements.length; i++)
                    {
                        workList.add(elements[i]);
                    }
                }
            }
```

```
      public Field[] getFields()
      {
        int size = stack.size();
        Field[] result = new Field[size];
        int j = 0;

        for (final IValue stackElem : stack)
            {
            result[j++] = new Field()
                {
                    public String getName()
                    {
                        return "element";
                    }

                    public IValue getValue()
                    {
                        return stackElem;
                    }
                };
            }

        return result;
      }

}
```

APPENDIX D

PROGRAMS FOR CASE STUDY - RESOURCE ALLOCATION/DEALLOCATION (MULTISET EXTENSION)

This appendix contains the following four code listings showing the BIR representation of the Resource Contention system and the java classes that implement the multiset extension.

a) ResourceContension.bir
b) MultisetModule.java
c) IMultisetValue.java
d) DefaultMultisetValue.java

a) ResourceContension.bir

```
system TestMultiset
{
      extension Multiset for bogor.multiset.MultisetModule
      {
            typedef type <'a>;

            expdef Multiset.type <'a> create <'a> ('a ...);

            expdef boolean isEmpty <'a> (Multiset.type <'a>);

            expdef 'a selectElement<'a>(Multiset.type<'a>);

            expdef int frequency<'a>(Multiset.type<'a>, 'a);

            actiondef add<'a>(Multiset.type<'a>, 'a);

            actiondef remove<'a>(Multiset.type<'a>, 'a);

      }

      enum ResourceState { FREE, IN_USE }

      record Resource { ResourceState state; }

      record Disk extends Resource { }
      record Display extends Resource { }

      Resource DISK;
      Resource DISPLAY;


      Multiset.type<Resource> resources;
      main thread MAIN()
      {
            loc loc0: live {}
                  do
                  {
                        DISK := new Resource;
                        DISPLAY := new Resource;
                        Disk_f:=0;
                        Display_f:=0;
                        resources := Multiset.create<Resource>
                        (DISPLAY, DISK,DISPLAY, DISK, DISK);
                        start Process();
                        start Process();

                  }
                  return;
      }

      thread Process()
      {
            loc loc0: live {}
                  invoke run()
                  return;
      }

      function run()
      {
            Resource resource;
            loc loc0: live { resource }
                  when !Multiset.isEmpty<Resource>(resources) do
                  {
                        resource:= Multiset.selectElement
                              <Resource>(resources);
                        Multiset.remove<Resource>
                              (resources, resource);

                  }
                  goto loc2;

            loc loc2: live { resource }
                  do
                  {
                        resource.state := ResourceState.IN_USE;
                  }
                  goto loc3;

            loc loc3: live { resource }
                  do
                  {
                        resource.state := ResourceState.FREE;
```

```
                        }
                        goto loc4;

                  loc loc4: live {}
                        do
                        {
                              Multiset.add<Resource>
                                    (resources, resource);
                        }
                        goto loc0;

            }
      }
```

b) MultisetModule.java

```
   package bogor.multiset;
   import java.util.Map;
   import java.util.Properties;
   import edu.ksu.cis.projects.bogor.IBogorConfiguration;
   import edu.ksu.cis.projects.bogor.ast.Action;
   import edu.ksu.cis.projects.bogor.ast.Node;
   import edu.ksu.cis.projects.bogor.module.DefaultMessageStore;
   import edu.ksu.cis.projects.bogor.module.IBacktrackingInfoFactory;
   import edu.ksu.cis.projects.bogor.module.IExpEvaluator;
   import edu.ksu.cis.projects.bogor.module.IExtArguments;
   import edu.ksu.cis.projects.bogor.module.IMessageStore;
   import edu.ksu.cis.projects.bogor.module.IModule;
   import edu.ksu.cis.projects.bogor.module.ISchedulingStrategist;
   import edu.ksu.cis.projects.bogor.module.ISchedulingStrategyContext;
   import edu.ksu.cis.projects.bogor.module.ISchedulingStrategyInfo;
   import edu.ksu.cis.projects.bogor.module.IValueFactory;
   import edu.ksu.cis.projects.bogor.backtrack.IActionBacktrackingInfo;
   import edu.projects.bogor.backtrack.ITransformationBacktrackingInfo;
   import edu.ksu.cis.projects.bogor.module.state.IState;
   import edu.ksu.projects.bogor.throwable.NullPointerBogorException;
   import edu.ksu.cis.projects.bogor.module.value.IIntValue;
   import edu.ksu.cis.projects.bogor.module.value.INullValue;
   import edu.ksu.cis.projects.bogor.module.value.IValue;
   import edu.ksu.cis.projects.bogor.type.NonPrimitiveExtType;
   import edu.ksu.cis.projects.bogor.type.Type;
   import edu.ksu.cis.projects.bogor.type.TypeFactory;

   public class MultisetModule implements IModule {

   //  ~ Instance variables

       protected TypeFactory tf;

       protected IExpEvaluator ee;

       protected IValueFactory vf;

       protected IBacktrackingInfoFactory bf;

       protected ISchedulingStrategist ss;
```

```
 public IMessageStore connect(IBogorConfiguration bc) {
     tf = bc.getSymbolTable().getTypeFactory();
     ee = bc.getExpEvaluator();
     ss = bc.getSchedulingStrategist();
     vf = bc.getValueFactory();
     bf = bc.getBacktrackingInfoFactory();

     return new DefaultMessageStore();
}

public String getCopyrightNotice() {
     return null;
}

public IMessageStore setOptions(String arg0, Properties arg1) {
     return new DefaultMessageStore();
}

public void dispose() {
     tf = null;
     ee = null;
     ss = null;
     vf = null;
     bf = null;
}

public IMultisetValue create(IExtArguments arg)
 {
     Type multisetType = arg.getExpType();

     //build value object to be returned
     IMultisetValue result = new DefaultMultisetValue(
         vf,
         (NonPrimitiveExtType) multisetType,
         vf.newReferenceId());

     // add the arguments to the set
     int size = arg.getArgumentCount();

     for (int i = 0; i < size; i++)
     {
         result.add(arg.getArgument(i));
     }

     return result;
 }

 public IIntValue frequency(IExtArguments arg)
 {
     //gets the set
     if (arg.getArgument(0) instanceof INullValue)
     {
         throw new NullPointerBogorException();
     }
     IMultisetValue multiset = (IMultisetValue)
arg.getArgument(0);
     //get the element to be added
     IValue element = arg.getArgument(1);


     //returns a boolean depending on the emptiness of the set
     return multiset.count(element);
 }

public IIntValue isEmpty(IExtArguments arg)
 {
     //gets the set
     if (arg.getArgument(0) instanceof INullValue)
     {
         throw new NullPointerBogorException();
     }
     IMultisetValue set = (IMultisetValue) arg.getArgument(0);


     //returns a boolean depending on the emptiness of the set
     return getBooleanValue(set.isEmpty());
 }

public IValue selectElement(IExtArguments arg)
 {
     //gets the elements of the set
     if (arg.getArgument(0) instanceof INullValue)
     {
         throw new NullPointerBogorException();
     }
     IMultisetValue multiset = (IMultisetValue)
arg.getArgument(0);
     IValue[] elements = multiset.elements();

     //ask the scheduler which one should be picked now
     int index = ss.advise
           (arg.getExtDesc(), arg.getNode(), elements, arg
         .getSchedulingStrategyInfo());

     //returns the one picked by the scheduler
     return elements[index];
 }

 protected IIntValue getBooleanValue(boolean b)
 {
     return vf.newIntValue(tf.getBooleanType(), b ? 1 : 0);
 }

 public IActionBacktrackingInfo add(IExtArguments arg)
 {
     //get the multiset
     if (arg.getArgument(0) instanceof INullValue)
     {
         throw new NullPointerBogorException();
     }
     IMultisetValue multiset = (IMultisetValue)
                               arg.getArgument(0);

     // get the element to be added
     IValue element = arg.getArgument(1);

     ISchedulingStrategyContext ssc=
                    arg.getSchedulingStrategyContext();

     //add the element
     multiset.add(element);

     //create the backtracking infos
     return new MultisetAdditionBacktrackingInfo(arg
             .getContainingTransition(),
              multiset, element, arg.getNode(), ssc
             .getStateId(), ssc.getThreadId(), arg
             .getSchedulingStrategyInfo());
 }

public IActionBacktrackingInfo remove(IExtArguments arg)
 {
     //get the set
     if (arg.getArgument(0) instanceof INullValue)
     {
         throw new NullPointerBogorException();
     }
     IMultisetValue multiset = (IMultisetValue)
arg.getArgument(0);

     //get the element to be removed
     IValue element = (IValue) arg.getArgument(1);

     ISchedulingStrategyContext ssc =
                              arg.getSchedulingStrategyContext();

     if (multiset.contains(element))
     {
         //remove the element
         multiset.remove(element);

         //create the backtracking information
         return new MultisetRemovalBacktrackingInfo(
             arg.getContainingTransition(),
             multiset,
             element,
             arg.getNode(),
             ssc.getStateId(),
             ssc.getThreadId(),
             arg.getSchedulingStrategyInfo());
     }
     else
     {
       //do nothing, so create no change backtracking information
         return bf.createNoChangeBacktrackingInfo(arg
             .getContainingTransition(),
                              (Action) arg.getNode(), arg
             .getSchedulingStrategyInfo());
     }
 }
 public static interface IMultisetAdditionBacktrackingInfo
 extends IActionBacktrackingInfo
{
}

public static interface IMultisetRemovalBacktrackingInfo
 extends IActionBacktrackingInfo
{
}

protected static class MultisetAdditionBacktrackingInfo
 implements IMultisetAdditionBacktrackingInfo
{
 ITransformationBacktrackingInfo parent;

 IMultisetValue multiset;

 IValue element;

 Node node;

 int stateId;

 int threadId;

 ISchedulingStrategyInfo ssi;

 /**
  * Public constructor
  */
 public MultisetAdditionBacktrackingInfo(
     final ITransformationBacktrackingInfo parent,
     final IMultisetValue multiset,
     final IValue element,
     final Node node,
     final int stateId,
     final int threadId,
     final ISchedulingStrategyInfo ssi)
 {
     this.parent = parent;
     this.multiset = multiset;
     this.element = element;
     this.node = node;
     this.stateId = stateId;
     this.threadId = threadId;
     this.ssi = ssi;
 }

 /**
  * Cloning constructor
  */
 private MultisetAdditionBacktrackingInfo()
 {
 }

 public ITransformationBacktrackingInfo getParent()
 {
     return parent;
 }

 public Node getNode()
 {
     return node;
 }

 public ISchedulingStrategyInfo getSchedulingStrategyInfo()
 {
     return ssi;
 }

 public int getStateId()
 {
     return stateId;
 }

 public int getThreadId()
 {
     return threadId;
 }

 public void backtrack(IState state)
 {
     multiset.remove(element);
 }

 public IActionBacktrackingInfo clone(Map<Object, Object>
cloneMap)
 {
     MultisetAdditionBacktrackingInfo bi =
(MultisetAdditionBacktrackingInfo) cloneMap
         .get(this);

     if (bi != null)
     {
         return bi;
     }

     bi = new MultisetAdditionBacktrackingInfo();
     cloneMap.put(this, bi);

     bi.element = element.clone(cloneMap);
     bi.node = node;
     bi.parent = parent.clone(cloneMap);
     bi.multiset = multiset.clone(cloneMap);
     bi.ssi = ssi.clone(cloneMap);
     bi.stateId = stateId;
     bi.threadId = threadId;

     return bi;
 }

 public void dispose()
 {
 }
}

protected static class MultisetRemovalBacktrackingInfo
 implements IMultisetRemovalBacktrackingInfo
{
 ITransformationBacktrackingInfo parent;

 IMultisetValue multiset;

 IValue element;

 Node node;

 int stateId;

 int threadId;

 ISchedulingStrategyInfo ssi;

 /**
  * Public constructor
  */
 public MultisetRemovalBacktrackingInfo(
     final ITransformationBacktrackingInfo parent,
     final IMultisetValue multiset,
     final IValue element,
     final Node node,
     final int stateId,
     final int threadId,
     final ISchedulingStrategyInfo ssi)
 {
     this.parent = parent;
     this.multiset = multiset;
     this.element = element;
     this.node = node;
     this.stateId = stateId;
     this.threadId = threadId;
     this.ssi = ssi;
 }

 /**
  * Cloning constructor
  */
 private MultisetRemovalBacktrackingInfo()
 {
 }

 public ITransformationBacktrackingInfo getParent()
 {
     return parent;
 }

 public Node getNode()
 {
     return node;
 }
```

```
        public ISchedulingStrategyInfo getSchedulingStrategyInfo()
        {
            return ssi;
        }

        public int getStateId()
        {
            return stateId;
        }

        public int getThreadId()
        {
            return threadId;
        }

        public void backtrack(IState state)
        {
            multiset.add(element);
        }

        public IActionBacktrackingInfo clone(Map<Object, Object>
       cloneMap)
        {
            MultisetRemovalBacktrackingInfo bi =
       (MultisetRemovalBacktrackingInfo) cloneMap
                .get(this);

            if (bi != null)
            {
                return bi;
            }

            bi = new MultisetRemovalBacktrackingInfo();
            cloneMap.put(this, bi);

            bi.element = element.clone(cloneMap);
            bi.node = node;
            bi.parent = parent.clone(cloneMap);
            bi.multiset = multiset.clone(cloneMap);
            bi.ssi = ssi.clone(cloneMap);
            bi.stateId = stateId;
            bi.threadId = threadId;

            return bi;
        }

        public void dispose()
        {
        }
    }
```

c) IMultisetValue.java

```
       package bogor.multiset;
```

```
import java.util.Map;
import edu.cis.projects.bogor.module.value.INonPrimitiveExtValue;
import edu.ksu.cis.projects.bogor.module.value.IIntValue;
import edu.ksu.cis.projects.bogor.module.value.IValue;
public interface IMultisetValue
extends INonPrimitiveExtValue {

   // add an element
    void add(IValue v);

    // determine whether a value is a member of this set
    boolean contains(IValue v);

    IValue[] elements();

    // determine whether this set is empty
    boolean isEmpty();

    //determine the frequency of the element
    IIntValue count(IValue v);

    // remove an element
    void remove(IValue v);

    // specialize return type of clone
    IMultisetValue clone(Map<Object, Object> cloneMap);
}
```

d) DefaultMultisetValue.java

```
package bogor.multiset;

import java.util.Arrays;
import java.util.Comparator;
import java.util.HashSet;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.Map;
import java.util.Iterator;
import java.util.Collection;
import java.util.Set;
import edu.ksu.cis.projects.bogor.IBogorConfiguration;
import edu.ksu.cis.projects.bogor.module.IValueFactory;
import
edu.ksu.cis.projects.bogor.module.value.INonPrimitiveValue;
import edu.ksu.cis.projects.bogor.module.value.IValue;
import edu.ksu.cis.projects.bogor.module.value.IIntValue;
import edu.ksu.cis.projects.bogor.module.value.IValueComparator;
import
edu.ksu.cis.projects.bogor.module.value.IValueVisitorAction;
import
edu.ksu.cis.projects.bogor.value.INonPrimitiveExtValue.Field;
import edu.ksu.cis.projects.bogor.type.NonPrimitiveExtType;
import edu.ksu.cis.projects.bogor.type.Type;
import edu.ksu.cis.projects.bogor.util.BitBuffer;
import edu.ksu.cis.projects.bogor.util.Util;
import edu.ksu.cis.projects.trove.custom.IntIntTable;
import edu.ksu.cis.projects.trove.custom.ObjectIntTable;

public class DefaultMultisetValue implements IMultisetValue
{
   protected IValueFactory vf;
   protected NonPrimitiveExtType type;
   protected int referenceId;
   protected Map<IValue,IIntValue> multiset = new
   HashMap<IValue,IIntValue>();

public DefaultMultisetValue(IValueFactory vf,
                  NonPrimitiveExtType type,int referenceId)
{
       this.vf = vf;
       this.type = type;
       this.referenceId = referenceId;
}
   // add an element
    public void add(IValue v)
    {
    if (!contains(v))
        {
            multiset.put(v,vf.newIntValue(1));
        }
        else
        {
            IIntValue val = multiset.get(v);
            int prev = val.getInteger();
            int cur = prev + 1;
            multiset.put(v, vf.newIntValue(cur));
        }

    }

    // determine whether a value is a member of this set
    public boolean contains(IValue v)
    {
       return multiset.containsKey(v);
    }

    //detemine the frequency
    public IIntValue count(IValue v)
    {

       if (!contains(v))
         {
             return vf.newIntValue(0);
         }
         IIntValue val = multiset.get(v);
         return val;
    }

    public IValue[] elements()
    {
    Set<IValue> set = multiset.keySet();
    IValue[] elements = set.toArray(new IValue[multiset.size()]);
```

```
 return elements;
}

 // determine whether this set is empty
 public boolean isEmpty()
 {
  return multiset.isEmpty();
 }

 // remove an element
 public void remove(IValue v)
 {
      IIntValue val = multiset.get(v);
      int prev = val.getInteger();
      int cur = prev - 1;
      if (cur >= 1)
      {
          multiset.put(v, vf.newIntValue(cur));
      }
      else
      {
          multiset.remove(v);
      }

 }

 public IMultisetValue clone(Map<Object, Object> cloneMap)
 {
     DefaultMultisetValue result =
            (DefaultMultisetValue) cloneMap.get(this);

     if (result != null)
     {
         return result;
     }

     Collection val = multiset.values();
     result = new DefaultMultisetValue(vf, type, referenceId);
     cloneMap.put(this, result);

     for (Iterator i=val.iterator(); i.hasNext( ); )
     {
       IValue element = (IValue)i.next( );
       result.add(element);
     }
     return result;
 }

 public void dispose()
 {
     if (multiset != null)
     {
     multiset.clear();
         multiset = null;
     }

     this.vf = null;
```

```
}

public int getReferenceId()
{
    return referenceId;
}

public Type getType()
{
    return type;
}

public int getTypeId()
{
    return type.getTypeId();
}

public byte[][] linearize(
        int bitsPerNonPrimitiveValue,
        ObjectIntTable<INonPrimitiveValue>
        nonPrimitiveValueIdMap,
        int bitsPerThreadId,
        IntIntTable threadOrderMap)
    {
        BitBuffer bb = new BitBuffer();

        IValue[] sortedElements = elements();

        vf.newVariedValueArray(sortedElements).linearize(
            false,
            bitsPerNonPrimitiveValue,
            nonPrimitiveValueIdMap,
            bitsPerThreadId,
            threadOrderMap,
            null,
            bb);

        return new byte[][]
            {
                bb.toByteArray()
            };
    }

public void visit(
        final IValueComparator vc,
        boolean depthFirst,
        Set<IValue> seen,
        LinkedList<IValue> workList,
        IValueVisitorAction vva)
    {
        IValue[] elements = elements();

        if (depthFirst)
        {
            for (int i = elements.length - 1; i >= 0; i--)
            {
                workList.addFirst(elements[i]);
            }
        }
        else
        {
            for (int i = 0; i < elements.length; i++)
            {
                workList.add(elements[i]);
            }
        }
    }

public void validate(IBogorConfiguration bc)
{
    type = (NonPrimitiveExtType) bc
        .getSymbolTable()
        .getTypeIdTypeTable()
        .get(type.getTypeId());

    vf = bc.getValueFactory();
}

public Field[] getFields()
{

Set<Map.Entry<IValue, IIntValue>> set1 = multiset.entrySet();
int size = set1.size();
     Field[] result = new Field[size];
     int j = 0;

     for (final Map.Entry<IValue, IIntValue> setElem : set1)
     {
         result[j++] = new Field()
             {
                 public String getName()
                 {
                     return "element";
                 }

                 public IValue getValue()
                 {
                     return setElem.getKey();
                 }
             };
     }

     return result;

}

public int hashCode()
{
    return getReferenceId();
}

public int compareTo(IValue o)
{
    if (o == null)
```

```
        {
            throw new NullPointerException();
        }

        // all IValue's are "less than" other objects
        if (!(o instanceof IValue))
        {
            return -1;
        }

        // compare based on type id
        int typeComp = Util.compare
              (getTypeId(), ((IValue) o).getTypeId());

        if (typeComp != 0)
        {
            return typeComp;
        }
        DefaultMultisetValue other = (DefaultMultisetValue) o;

        return Util.compare(getReferenceId(),
other.getReferenceId());
    }
}
```

VITA

Minal Wad

Candidate for the Degree of

Master of Science

Thesis: EXPLORING DOMAIN SPECIFIC APPROACHES TO SOFTWARE MODEL CHECKING

Major Field: Computer Science

Biographical:

Personal Data: Born in India on June 8, 1981, daughter of Vishweshwar Mahadeo Wad and Pournima Wad.

Education: Received Bachelor of Science Degree in Computer Engineering from Mumbai University in June 2004; completed the requirements for the Degree of Master of Science at the Computer Science Department at Oklahoma State University in December 2006.

Experience: Employed at Motorola, Inc. as an intern from May 2005 to August 2005 and from May 2006 to August 2006; graduate research assistant and teaching assistant at the Computer Science Department of Oklahoma State University from August 2004 to December 2006.

Name: Minal Wad Date of Degree: December 2006

Institution: Oklahoma State University Location: Stillwater, Oklahoma

Title of Study: EXPLORING DOMAIN SPECIFIC APPROACHES TO SOFTWARE MODEL CHECKING

Pages in Study: 82 Candidate for the Degree of Master of Science

Major Field: Computer Science

Scope and Method of Study: Model checking has proven to be an effective technology for verification and debugging in hardware domains and more recently in software domains. The major challenges in the application of model checking to software systems are: the mapping of software executables to model checker's input language and the intrinsic complexity of the ever growing software systems. This thesis explores the domain specific model checking approaches to large systems in order to optimize the state space storage for specific domains. Bogor [Bogor 2003] is an extensible, customizable, and highly modular model checking framework that supports general as well as domain specific software model checking. As a part of the thesis, domain specific extensions to Bogor's input language, called Bandera Intermediate Representation (BIR), were implemented by providing a plugin for Eclipse [Eclipse 2004]. Eclipse is a universal platform for tool integration and its plugin development environment facilitates addition of new plugins to the existing ones. Eclipse's extension mechanism is exploited by Bogor. Bogor was installed as an Eclipse plugin and with the help of Eclipse's Plugin Development Environment (PDE), new data types were integrated with the existing Bogor framework.

Findings and Conclusions: Two case studies ('postfix calculator' using stack extension and 'resource allocation' using multiset extension) were investigated. Various metrics such as number of states, transitions, and maximum depth were analyzed. The complexity of the test cases was increased gradually to test the extensions for feasibility and scalability. The thesis also involves a comprehensive study of some of the well-known model checkers and their features, degree of automation, and input languages. It was observed that customizing the model checker as per domain specifications helped in achieving space reduction. The space reduction is prominent, especially in large domains where it contributes towards state space explosion solution. Although development of extensions is achievable, it requires a working knowledge of Eclipse and specific knowledge of model checking. In conclusion, a domain specific approach for software model checking was demonstrated to be a promising technology. Language extensions to BIR were successfully built and tested for accuracy and scalability.

ADVISER'S APPROVAL: M. H. Samadzadeh