Enforcing Resource-Aware Policies Using Run-time Monitoring

by

Natalie Alexandra Bowles

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen's University
Kingston, Ontario, Canada
January 2004

Abstract

Resource awareness is an important step towards the realization of reliable and effective communication systems, particularly in the context of mobile code. Since the amount of consumption of computing resources (file system, threads, sockets, etc.) is not monitored in most mobile execution environments such as in the Java language, some kind of change to the security model seems indispensable.

It is the goal of this research to gain insight into a richer security model – one that enables monitoring of resource consumption and ensures that allowable constraints are met. Specifically, we extend an existing approach to monitoring and checking of running Java programs to the enforcement of resource-aware policies. Specifically, our work focusses on monitoring and controlling aspects of file system use.

Acknowledgments

I would like to express deep appreciation to my supervisor, Prof. Juergen Dingel, for his guidance and support in this research project. I would also like to thank Prof. Oleg Sokolsky and Usa Sammapun of the University of Pennsylvania for their helpful and responsive suggestions.

I am indebted to the Office of Critical Infrastructure Protection and Emergency Preparedness (OCIPEP) of the Government of Canada for their generous financial support.

I am deeply grateful to my parents, Anna and Alan, and my brother Paul, for their love and support.

Thank you.

Finally, I would like to dedicate this thesis in memory of my Babyshka, Alexandra Alekov.

Contents

Abstract **i**

Acknowledgments **ii**

Contents **iii**

List of Tables **vi**

List of Figures **vii**

1 Introduction **1**

1.1 Motivation 1

1.2 Problem and Approach 3

1.3 Objective 3

1.4 Contributions 4

1.5 Organization of Thesis 5

2 Background **7**

2.1 Introduction to Java Security 8

2.1.1 Existing Permission Syntax and Enforcement 10

2.1.2 Java API 13

2.1.3 Security Manager and the Java API 13

2.1.4 Applet Restrictions and Capabilities 14

2.1.5 Conclusion 16

2.2 Review of the Literature 17

2.2.1 Modifying the JVM 17

2.2.2 Extending Java Native and System Code 17

2.2.3 Program Transformations 21

2.2.4 Conclusion 23

3 Overview of MaC **25**
3.1 Introduction to MaC 25
3.2 Bytecode Instrumentation 26
3.3 Overview of the MaC Architecture 27
3.3.1 Static Phase of the MaC Architecture 28
3.3.2 Run-time Phase of the MaC Architecture 29
3.4 The MaC Languages 29
3.4.1 Primitive Event Declaration Language (PEDL) 30
3.4.2 Meta Event Declaration Language (MEDL) 32
3.4.3 Example 34
3.5 Summary 36
3.6 Application of MaC to Monitoring and Control of Resource Consumption 37

4 Extensions **39**
4.1 Monitor API Methods 42
4.2 Monitor File System Use 49
4.2.1 Allow Method Arguments to be Communicated 51
4.2.2 Allow Strings to be Communicated 57
4.2.3 Add Sets to MEDL 63

5 Examples **67**
5.1 Small Examples 67
5.1.1 Example: Limit Read/Write 68
5.1.2 Example: Print Filenames 70
5.1.3 Example: Limit Amount of File Access 71
5.1.4 Example: Restrict Access to Particular File 72
5.1.5 Example: Restict File Access Amount 74

6 Summary and Future Work **77**
6.1 Summary 77
6.2 Future Work 78
6.2.1 Short Term 78
6.2.2 Future Work 79
6.2.3 Alternative Modes of Communication 79
6.2.4 Logical Frameworks Expansion 80
6.2.5 Feedback Capability Extension 81
6.2.6 Application to other Languages 81

Bibliography **82**

A BNF for PEDL **87**

B BNF for MEDL **91**

Vita **95**

List of Tables

2.1 Example Policy File 11
2.2 Example File Permission Syntax 11

3.1 The Syntax of Conditions and Events 30
3.2 Structure of a MEDL Specification 33

List of Figures

3.1 Overview of the MaC Architecture 27
3.2 Excerpt of the PEDL Specification for the Gate Controller 35
3.3 Excerpt of the MEDL Specification for the Gate Controller 36

4.1 Excerpt of Subclass `BufferedReader.java` 44
4.2 Excerpt of PEDL Specification for Subclass `FileInputStream.java` and `BufferedReader.java` 45
4.3 Excerpt of MEDL Script for Example Allow Method Arguments to be Communicated. 52
4.4 Example of Snapshots from Filter to Event Recognizer in Original MaC 55
4.5 Example of Snapshots from Filter to Event Recognizer After Our Extension Allow Method Arguments to be Communicated 56
4.6 Excerpt of PEDL Script for Example Allow Strings to be Communicated. 59
4.7 Excerpt of MEDL Script for Example Allow Strings to be Communicated. 59
4.8 Example of Snapshots from Filter to Event After Our Extension Allow Strings to be Communicated 62
4.9 Excerpt of MEDL Script for Example Add Sets to MEDL. 64

5.1 Excerpt of PEDL Script for Example Limit Read/Write. 69
5.2 Excerpt of MEDL Script for Example Limit Read/Write. 69
5.3 Excerpt of MEDL Script for Example Print Filenames. 71
5.4 Excerpt of MEDL Script for Example Limit Amount of File Access. . 72
5.5 Excerpt of PEDL Script for Example Restrict Access to Particular File. 73
5.6 Excerpt of MEDL Script for Example Restrict Access to Particular File. 73
5.7 Excerpt of MEDL Script for Example Restrict File Access Amount. . 74

Chapter 1

Introduction

1.1 Motivation

As the use of bytecode as a medium of exchange becomes integrated into our daily lives, our dependence on security increases. Electronic communication for the military, business and the public at large depend on a protected technology infrastructure [Mar03].

Resource awareness in the context of mobile code is an important concept for application security. Hosts that execute unknown foreign code have to control resource consumption: the system has to account for resources consumed by mobile code and prohibit allocations that exceed foreign code resource limits. This way, applet attacks which may limit access to a program by legitimate users that are launched by malicious (or buggy) segments of foreign code, are prevented [VB01].

The popularity of the Web has lead to more regular use of Java and other languages that support mobility. Together with increased security threats posed by importing more and more mobile code, this development has caused an emphasis on the security of executing Java applets [MS98]. Mobile code is risky: What might it do? What if it is hostile? The naive user moves from one Web page to another, unaware that Java applets and other forms of mobile code are included in Web pages, ready to be executed on the page reader's machine [Mar03].

How can users protect against possible resource-consuming attack? The Java Virtual Machine (JVM), an abstract computing machine and execution environment for the Java language, includes a bytecode verifier that checks bytecode properties before execution and a bytecode interpreter that performs run-time tests [CMS01]. Despite the support for initial access control, the Java sandbox lacks control over the *extent* to which a resource can be used. The sandbox model features a restrictive environment in which to run untrusted code. Once access to a resource has been granted to a piece of code, the code is free to use it to any extent. Therefore, control is necessary to prevent resource-consuming attacks: control that imposes limits on privileges and address the availability of resources. Otherwise, excessive amounts of data could be written to disk for instance, denying other applications access to the file system because the target system is overwhelmed. Many variants of this kind of attack exist [HS02b].

A mechanism that monitors and controls resource usage is needed. In this work, we develop a mechanism that monitors and controls aspects of Java file system resource usage. Our approach relies on a portable transformation technique applied directly to executable Java bytecode; we demonstrate that it is possible to use run-time monitoring to enforce resource-aware policies.

1.2 Problem and Approach

While the Java Virtual Machine includes a bytecode verifier that checks bytecode programs before execution, and a bytecode interpreter that performs tests at run-time, mobile code may still behave in ways that are harmful to users [CMS01]. Java's security architecture is known for not taking resource availability into account. More precisely, Java lacks the ability to monitor the *extent* of resource (file system, threads, sockets, etc.) use during a program's execution [HS02b].

This thesis intends to provide a mechanism for monitoring resource consumption of mobile programs. Our work develops a tool for monitoring and checking mobile code in a flexible, extensible and fine-grained manner. Firstly, we provide a mechanism to create flexible and configurable resource-aware policies for mobile code, by means of our extensions of MaC, a run-time monitoring and checking tool. In addition, we monitor resource requirements described in a formal language. This enables the user to customize his or her version of personally designed security policies to place constraints on how an applet uses system resources.

1.3 Objective

The goal of this research is to examine the application of run-time monitoring to the enforcement of resource-aware policies. Specifically, we will be looking at how to produce a resource controlled system that could safely execute untrusted Java bytecode. Our work focuses on aspects of file system resources.

To leverage existing work, we chose to extend a tool called Monitoring and Checking (MaC), which monitors a running system and checks its run-time behaviour against a formal requirement specification. In its original form, MaC provides a framework that offers fine-grained control over policy enforcement, simple composition of policy specification, and a flexible mechanism for user customization, in a Java run-time environment. We extend MaC to allow it to enforce resource-aware security policies. Our extension focusses on aspects of file system resources.

1.4 Contributions

This thesis makes the following contributions:

1. We designed and implemented an extension of MaC to specify and enforce a class of resource-aware security policies
2. Our work adds the following functionalities to MaC:
    - Ability to monitor Java API methods
    - Ability to communicate method arguments
    - Ability to communicate Strings
    - Ability to use sets in MEDL (the requirements specification language of MaC)
    - Perform set operations `add, emptySet, size,` and `print`
3. We evaluated our work on several small examples

Note that the extensions listed in 2. are useful using MaC for run-time monitoring in general – even when security or resource usage is not of concern.

1.5 Organization of Thesis

The organization of this thesis is as follows:

- *Chapter 1:* This chapter provides the introduction of the thesis.
- *Chapter 2:* We will provide background in the area of Java Security and provide a review of the literature of existing work on methods to enforce resource control in Java environments.
- *Chapter 3:* We will present a tool called Monitoring and Checking (MaC), which monitors a running system and checks its run-time behaviour against a formal requirement specification. In our work, we extend MaC to provide a mechanism that monitors and checks aspects of file resource consumption of applets.
- *Chapter 4:* This chapter is the main section of this thesis. We will describe our extensions to the MaC tool necessary for monitoring file system use. We provide examples and implementation details and describe the limitations.
- *Chapter 5:* We will provide some small examples of resource-aware policies that were not possible before, but can now be specified and enforced with MaC, using the extensions this thesis has contributed.
- *Chapter 6:* This is the concluding chapter of the thesis. In this chapter, the thesis is summarized and areas of future work are discussed.

We also provide appendices to complement the material presented in the chapters of this thesis:

- *Appendix A,B:* We provide the grammar in Backus-Naur Form (BNF) for the Primitive Event Declaration Language (PEDL) and Meta-Event Declaration Language (MEDL) specifications, respectively. PEDL and MEDL are used by MaC to express the constraints that are to be enforced on the monitored system. PEDL and MEDL consist of the monitor and requirements specifications of the MaC language, respectively.

Chapter 2

Background

In this chapter, we review the Java security architecture and arrive at an understanding of what security is provided by the architecture that addresses resource control. This understanding is the foundation for our proposed solution for secure resource control given in Chapter 4. In Chapter 4 we present our extensions to a tool called Monitoring and Checking (MaC), which monitors a running system and checks its run-time behaviour against a formal requirement specification. Our extension allows MaC to enforce a limited class of resource-aware security policies.

The purpose of Section 2.1 is to provide a high-level introduction to Java security. Section 2.1.1 to Section 2.1.4 delve into deeper details: Section 2.1.1 provides some background on the existing Java permission syntax and enforcement, which helps to understand the necessity of extending the resource security features of Java. Section 2.1.2 explains Java's use of its core class library (API). Section 2.1.3 extends the discussion of the Java API in context of the Java Security Manager. Section 2.1.4 discusses the security restrictions and capabilities most applets face. Section 2.1.5 concludes our introduction to Java security.

Section 2.2 reviews existing work on methods to enforce resource control in Java environments. We describe the goals of these methods and the approaches these methods have taken. Prevailing approaches fall under one of three main categories: Section 2.2.1 to Section 2.2.3 concern approaches that involve modifying the JVM, extending Java native and system code, and program transformations, respectively.

Section 2.2.4 concludes our review of the literature. In this section we also briefly introduce how the development of MaC could provide better security than the current Java run-time execution environment affords. This can be achieved by monitoring resource consumption and assurance that allowable constraints are met. Our research extends the assurance the run-time monitoring tool MaC brings with resource-awareness, to prevent the risk of a particular class of resource-consuming attacks.

2.1 Introduction to Java Security

Since its birth in the early 1970s as a 12-node network called the ARPANET, the Internet has expanded into a network that forms an important part of the world's information infrastructure. Initially, the Internet was almost completely text-based: It was used to transfer files and to communicate via email. Further developments made it possible to send mobile code across the Net. Mobile code is based on the idea of sending code to a remote computer to be executed. Now, all current Web browsers include the capability of running mobile code automatically [MF99].

Java brings the ability to add mobile code to Web pages, independent of the client's platform [CMS01]. McGraw and Felten explain the way Java works in this context: The administrator must first install a Java Virtual Machine (JVM), or a browser that includes a JVM [MF99]. Then, a requesting Web browser (1) fetches the code from the Web, (2) automatically verifies it before the code is run, and (3) performs additional tests at run-time.

A fundamental concern lies in the fact that Java code is automatically downloaded across the network and runs on the client's machine. This means that a networked computer can import and execute Java mobile code (bytecode) in ways that are invisible or only partly visible to the user. For example, a user (or browser) may execute a Java program embedded within a Web page. This mobile code that traverses a network and executes at a remote site can behave in ways that are annoying, erroneous, and/or potentially harmful – and it runs in the same environment as all of the machine's resources [MS98, CMS01].

Any mobile code platform, including Java, suffers from four basic categories of mobile code risks: leakage, tampering, resource stealing, and antagonism **??**.

Leakage

In this case, unauthorized access is attempted, to obtain information belonging to or intended for someone else.

Tampering

Unauthorized change or deletion of information is called tampering.

Resource Stealing

This occurs when there is unauthorized use of computing resources or facilities such as memory or disk space.

Antagonism

These are interactions that don't result in a gain for the intruder, but are annoying for the attacked party.

To deal with these threats, the Java run-time environment provides a security model that tries to protect users from erroneous or malicious code.

Java provides fairly good levels of protection against leakage and tampering, but resource stealing and antagonism cannot be fully prevented since it is difficult to distinguish between legitimate and malicious actions [Ais03]. Attacks that prevent someone from using his or her machine are called denial-of-service attacks in the security community. McGraw and Felten explain that denial-of-service attacks are by far the most commonly encountered Java Security concern and go on to say:

> "Implementing such an attack is not hard, but stopping one is" [MF99].

Denial-of-service attacks can come in many different forms. For instance, a brute-force attack may simply attempt to use all available resources. A more sophisticated attack may be based on legitimate, but taxing use of resources. Unfortunately, Java's security architecture does not take resource consumption into account.

2.1.1 Existing Permission Syntax and Enforcement

The Java Security model implements a "sandbox" that imposes strict controls over what a Java program can and cannot do. The sandbox model features a restrictive environment in which to run untrusted code. These controls are based on a policy and an enforcement mechanism [HS02b].

Herzog and Shahmehri explain existing Java permissions, their syntax, and the existing enforcement mechanism [HS02b]. A policy consists of a sets of permissions.

```
keystore "file:/home/user/michael/.keystore/";
grant signedBy "Kathryn" {
  permission java.security.AllPermission;
};
grant signedBy "Andrew", codeBase "file:/sw/li/tool.jar" {
  permission java.io.FilePermission "/tmp/a", "read";
}
grant codeBase "file:/sw/lib/app.jar" {
  permission java.io.FilePermission "/tmp/b", "read";
};
```

Table 2.1: Example Policy File

permissions java.io.FilePermission(“/tmp/myLogFile”,“read,write,execute,delete”)

Table 2.2: Example File Permission Syntax

A set of permissions is assigned to a given code base and/or a signer. Consider, for instance, the example policy file given in Table 2.1. It contains the policy of a user called Michael. As illustrated in Table 2.1, Michael trusts code that was signed by Kathryn with full permissions. Michael allows Andrew's tools only to read the file `/tmp/a`. Code in the Java archive (Jar) file `app.jar` is only allowed to read from `/tmp/b` [HS02b].

A Java permission is identified by its class name, a target string, and an action string. The example file permission syntax given in Table 2.2 illustrates how file permissions are identified by their class name `java.io.FilePermission`. The target string in a Java permission identifies the file(s) to which the permission applies, and the action string denotes which actions are allowed for the given file(s) [HS02b].

The policy is enforced by Java's Security Manager. The Security Manager is a single Java object that performs run-time checks on security-sensitive methods. The Java core classes (API) provide all calls necessary to interface with operating system resources. Whenever a dangerous call is made to the Java library, the library queries the Security Manager [HS02b].

The Java library's use of the Security Manager works as follows [Her02]:

1. A Java program makes a call to a potentially security-sensitive operation in the Java API.

2. The Java API code asks the Security Manager whether the operation should be allowed. Permissions are checked against the rules specified in the policy file.

   A key question is how the Security Manager decides whether access to a resource is granted. To decide whether access to a resource is granted, a stack inspection algorithm and a security policy are used. The algorithm searches stack frames on the caller's stack in order from newest to oldest. If the search finds a stack frame with the appropriate enabled-privilege annotation, it terminates, allowing access. If the search finds a stack frame that is forbidden from accessing the target by local policy, or has explicitly disabled its privileges, the search terminates, forbidding access [MF99].

3. The Security Manager throws a SecurityException back to the Java API if the operation is denied. This exception propagates back to the Java program.

4. If the operation is permitted, the call to the Security Manager returns without throwing an exception, and the Java API performs the requested dangerous operation and returns normally.

2.1.2 Java API

Java applications use the Java Application Program Interface (API) of the Java Development Kit (JDK) [Yak02]. Java applications access system resources (such as I/O, for example) by calling methods in the classes that implement the Java API. In addition, the Java API contains numerous native methods, that is, methods that are written in a different language than Java. Native methods can be used to provide Java programs with direct access to the resources of the underlying operating system.

JDK classes are tightly interwoven with these native methods [Yak02]. These dependencies, which are not well documented, have to be respected in order to preserve the integrity of the Java platform. For instance, it is important that the structure of the call stack, when a native JDK method is invoked, remains intact. It is therefore very difficult to change the JDK classes without violating these dependencies. Consequently, the opportunities to enforce security constraints by changing the Java API are very limited.

2.1.3 Security Manager and the Java API

The Security Manager defines the outer boundaries of the "sandbox". The sandbox model features a restrictive environment in which to run untrusted code. A Security Manager is any class that descends from class `java.lang.SecurityManager`. The Security Manager is customizable. The Java API enforces the custom security policy by asking the Security Manager for permission to take any action, before it fulfills requested activity on a system resource [Ven97].

For each security-sensitive operation, there is a method in the Security Manager that defines whether or not that action is allowed by the sandbox. Each such method's name starts with “check”, so for example `checkRead()` defines whether or not a thread is allowed to read from a specified file, and `checkWrite()` defines whether or not a thread is allowed to write to a specified file. The implementation of these methods is what defines the custom security policy of the application [Ven97].

Note that only Java applications are considered “trusted” and can install a custom Security Manager. By default, a Java application has no Security Manager. It can choose to install one. However, for security reasons Java applets that are loaded over the network cannot install a new Security Manager. Applets are subject to the Security Manager of the application (Web browser or appletviewer) in which they are running [CW96].

2.1.4 Applet Restrictions and Capabilities

Campione and Walrath [CW96] describe the security restrictions and capabilities of applets. Applet restrictions include:

1. An applet cannot load libraries or define native methods.
2. An applet cannot read or write files on the host that is executing it.
3. An applet cannot make network connections except to the host that it came from.
4. An applet cannot start any program on the host that is executing it.
5. An applet cannot read the system properties: `user.name`, `user.home`, `user.dir`, `java.home`, and `java.class.path`.

6. Windows that an applet brings up look different than windows that an application brings up.

Besides the obvious feature that applets can be loaded over the network, applets also have the following capabilities:

1. An applet can play sounds.
2. An applet running within a Web browser can cause HTML documents to be displayed and control in which browser window(s).
3. An applet can invoke public methods of other applets on the same page.
4. An applet that is loaded from the local file system (from a directory in the user's classpath) have none of the restrictions that an applet loaded over the network does. This is because applets that are in the user's classpath become part of the application when they are loaded.
5. Although most applets stop running once their page is left, they do not have to. Java does not require that applets implement the `stop()` method (if necessary) to stop any processing when the user leaves the applet's page.

In short, applets that are loaded over the network come from a possible unknown source with unknown proper functioning, and run in the same environment as the client's computer system resources. Most applets "should" not abuse their use of computing resources, but it is not outside their capabilities [CW96].

2.1.5 Conclusion

Despite the support for initial access control, the Java sandbox does not allow control over the extent to which a resource can be used. The policy for a Java application environment, specifying which permissions are available for code from various sources, only defines rules for actions particular pieces of code are allowed to perform. Access is granted or forbidden without further qualifications. In other words, once access to a resource has been granted to a piece of code, the code is free to use it to any extent. For example, with permission granted to write in directory `/tmp`, there is no control over, for instance, how many bytes are written or how many files are created. Malicious or otherwise malformed code could exceed its allotment of resource use and make system resources unavailable.

Secondly, the JDK classes do not lend themselves easily to instrumentation. The tight interweaving of JDK classes causes subtle complications for the rewriting of JDK classes. Unfortunately, monitoring the use of the Java API is necessary for an accurate awareness of resource utilization.

Lastly, applet restrictions and capabilities motivate the need for run-time monitoring of resource consumption. By default, applets are subject to strict restrictions of their access to system resources. Security policies can be customized somewhat using the policy files read by the Security Manager. However, creation of more fine-grained policies is difficult and error-prone because it requires the modification of the checking routines. Our extension of run-time monitoring and checking will allow for the easy creation and enforcement of fine-grained, abstract security policies.

2.2 Review of the Literature

The focus of this review of the literature is on previous work that has been done to bring resource awareness to Java. Prevailing approaches rely either on modifying the JVM, extending Java native and system code, or on program transformations [HK02].

2.2.1 Modifying the JVM

Approaches that modify the JVM offer a replacement for the standard Java run-time system environment. NOMADS is a mobile agent system, based on a modified JVM that supports resource control [SBB$^+$00]. The NOMADS environment is composed of two parts: an agent execution environment called Oasis and a new Java compatible Virtual Machine called Aroma.

Unfortunately, since NOMADS is based on a modified JVM, its portability is limited. Also, the configurability of this approach is questionable: NOMADS is based on a set of standard Java security policies and mechanisms, and as such is not easily amenable to user customization [VB01, SBB$^+$00].

2.2.2 Extending Java Native and System Code

Extending Java with resource management support involves the modification of Java system and/or native code libraries. Both require expression in a given programming language. Furthermore, modification of Java native code libraries is inherently different from language run-time system customization: Java API classes are tightly interwoven with native code of the Java run-time system. Dependencies between Java native and system code are not well documented [Yak02].

JRes [CvE98]

JRes is a resource control library for Java, introduced by Czajkowski and von Eicken [CvE98]. The JRes prototype is implemented on top of standard JVMs and requires a small amount of native code.

JRes allows accounting for CPU time, heap memory, and network resources. The basic idea is to add resource accounting and limiting facilities to Java, as a class library that replaces other core Java classes by bytecode rewriting. JRes introduces a Resource Manager class that co-exists with the Java class loader and the Security Manager [Yak02].

Accounting for CPU time relies on a mixture of bytecode rewriting and native code. During thread registration, a native code routine is invoked to create a new thread handle. The new handle is rewritten to query the operating system for CPU consumption [CvE98].

The goal of memory accounting is to know at any point in a program's execution, how much memory is used. Appropriate bytecode is inserted whenever an object allocating instruction occurs in the original method code. Further, methods which allocate memory will change during bytecode transformation: JRes needs the support of a native method to account for memory occupied by array objects [CvE98].

To achieve accounting for network resources, a new `java.net` package is inserted. The `java.net` package comprises the core classes Java uses to access the network directly [CvE98].

JRes extends Java with an interface that is flexible to allow resource consumption policies to be built. It motivates the incorporation of a resource management interface into the language specification of Java. However, JRes is not easy to use. It requires transformation of native code, which in itself requires knowledge and skill in Java and manipulation of its core class libraries [CvE98]. Furthermore, with the introduction of a resource manager class into the existing security architecture, there are a number of restrictions on what it can do [Yak02].

Herzog and Shahmehri [HS02b]

Herzog and Shahmehri implement resource control by extending the existing Java permission syntax for stating policies [HS02b]. Recall, a Java permission is identified by its class name, a target string, and an action string. The target string in a Java permission identifies the file(s) to which the permission applies. In the work of Herzog and Shahmehri, the action string denotes which actions are allowed for the given file(s). The target string can be extended with a target-specific resource limit

$$< target > [:< global\_target\_limit >]$$

that restricts access to the resource as a whole. The action string, if present, can receive an action-specific resource limit:

$$< action > [:< action\_limit >] \ .$$

For handling purposes, an abstract permission class is built that extends

```
java.security.Permission .
```

With this approach, the abstract class which originally defined the set of standard permissions for accessing a system resource, can then be subclassed for specific permissions [HS02b].

This work provides integration of resource control into the existing Java security architecture. This contributes to a clear design of the security framework. The disadvantage of this solution is that it adds complexity to the already complex Java security architecture. Extension of the target and action strings allow target- and action-specific resource limits to be appended onto the existing permission syntax. In addition, ways of achieving this solution require modification of the Java virtual machine (JVM) or automatic class modification at run-time. This work is only a proof-of-concept. In order to work with it, ways must be found to force the use of its resource-aware classes [HS02b].

Mehta and Sollins [MS98]

Mehta and Sollins expanded the security features of Java by modifying the Security Manager, rather than the JVM or the system classes [MS98]. In this work, they built a constraint language to describe more complex permission constraints within the scopes of the sandbox model. In addition, logging facilities were added to the Java security architecture. A log is maintained to keep track of the actions by applets.

The Security Manager was extended through modification of several `checkX` methods. Originally, `checkX` methods were used when the applet's Security Manager was questioned by the Java system classes upon request for access to a system resource. In this work, when one of the Security Manager's `checkX` methods is called, it uses the rules and the logs to determine whether the permission is granted.

This approach introduces a policy language that allows specification of constraints to go beyond the simple sandbox model of identity and actions, by means of logging facilities that track applets as file owners and their accesses. The disadvantage in this approach is that it requires the user to know how to modify the Security Manager.

Several `checkX` methods in the applet Security Manager must be modified, because these methods are used to check access and their activity must be logged [MS98].

2.2.3 Program Transformations

Approaches that rely on program transformation perform bytecode transformation: Executable code is rewritten by inserting additional code [Yak02].

J-RAF [Yak02]

The Java Resource Accounting Framework (J-RAF) is exclusively based on program transformation [HK02, Yak02]. A resource control layer is added to the bytecode of Java programs. In this approach, the bytecode of applications is rewritten in order to make their CPU and memory consumption explicit. Programs rewritten with J-RAF keep track of the number of executed bytecode instructions and update a memory account interface that accounts for the size, in bytes, of the classes loaded and instantiated objects created by the application.

J-RAF understands that rewriting the bytecode of an application is not sufficient to account for and control its resource consumption, because Java applications use the APIs of the Java Development Kit (JDK). Because native code may invoke Java methods and J-RAF does not modify native code, this approach adds wrapper methods which access the account objects associated with each thread, and passes them to the resource-aware methods that take these account objects as extra arguments. However, significant complexity is added to the rewritten API class. Usage monitoring and resource control instructions are inserted into the code, increasing the size of the code by about 80% for the different JDKs [Yak02].

Chander, Michell, and Shin [CMS01]

Chander, Mitchell, and Shin introduce bytecode instrumentation to monitor and control resource usage [CMS01]. New, safe classes are substituted to replace the original Java classes before bytecode execution. Their technique is performed as follows:

1. Identify the Java API classes which control the resource the user is interested in monitoring.
2. Subclass these API classes to create a safe version of the original class, which contains user defined checks.
3. Invoke the bytecode filter on the original API class, so that all references to the original class are replaced with references to the new (safe) subclass.

Handling final classes requires bytecode modification at the method-level, but follows a similar approach [HS02b].

Their system captures and modifies Java bytecode before it enters the browser by means of a piece of software called a "network proxy". When a Web server sends a Java applet, the proxy will pass the applet code to a bytecode filter. The bytecode filter will examine the bytecode for potential risks, and modify the bytecode before sending it for execution to the Web browser. The proxy also has access to a repository of Java classes, including safe library classes that can be substituted for original library classes, if need be. Safe library classes restrict resource usage and functionality. Unfortunately, the technique provided by Chander, Mitchell, and Shin [CMS01] is restricted to a standard definition of resource constraints. Alternatively, a "security-tuner" interface can be installed, that allows the user to customize their security constraints. The portability of this alternative is limited.

JPaX [Hs01]

Close in spirit with our work with MaC, Havelund and Roşu present Java PathExplorer (JPaX), a runtime verification system for monitoring Java programs [Hs01]. This tool facilitates automatic instrumentation of a program's bytecode, which will emit events to an observer during its execution, who will then check the events against user provided high-level requirement specifications.

Inspired by the MaC language framework, specification is split into an instrumentation script and a verification script. JPaX monitors the execution of temporal logic formulae, checking events against high-level requirement specifications, and error pattern analysis of deadlocks and data races, searching for low-level programming errors [Hs01].

This project mainly focuses on application to survey safety-critical systems. In an example analysis provided only integer variables are accepted by their specification language. Furthermore, no examples of policy statements are provided [Hs01].

2.2.4 Conclusion

Proposed solutions for resource control in Java rely either on a modified JVM, on extending Java native and system code, or on program transformations. Approaches that modify the JVM have limited portability. Extending the Java system and native code comes at a price of lower ease-of-use, flexibility, and configurability, because each requires knowledge of a given programming language. Existing approaches that use program transformations lack capability for application to monitor aspects of resource use and/or provide no example of their policy expression.

In Chapter 3, we present the development of Monitoring and Checking (MaC), a tool for monitoring systems and checking their run-time behaviour against a formal requirement specification at run-time. The requirement specification is unambiguously described in a formal language, defined for monitoring and checking purposes.

The availability of a monitoring component to protect against possible resource-consuming attacks should be considered crucial for any mobile code environment, in order to observe the behaviour of foreign code at run-time and to enforce user-defined policies for resource usage of mobile code.

Chapter 3

Overview of the Monitoring and Checking (MaC) Architecture

This chapter introduces the Monitoring and Checking (MaC) architecture, a framework that provides assurance on the correctness of program execution at run-time [Kim99]. Section 3.1 provides a brief introduction to MaC. Section 3.2 introduces the bytecode instrumentation tool that MaC uses. Section 3.3 gives an overview of the MaC architecture. Section 3.4 presents the monitoring and requirement specification languages of the MaC architecture. Section 3.5 summarizes this chapter's overview of MaC. Section 3.6 will clarify our choice of the MaC approach as the framework we will extend to enforce resource-aware policies.

3.1 Introduction to MaC

The MaC system was developed by the Real-Time systems Group (RTG) at the University of Pennsylvania, to provide a run-time formal analysis architecture. The approach is based on a framework of monitoring and checking a running system with the aim of ensuring that it is running correctly with respect to a formal specification of system requirements. The MaC approach involves instrumentation of a target program's executable codes (bytecodes), and checks the implementation against requirements at run-time [Kim99, KKLS01].

3.2 Bytecode Instrumentation

The Java bytecode instrumentation is performed using the JTrek bytecode engineering tool developed at Compaq [Hs01]. JTrek makes it possible to write Java applications that examine and modify Java class files. The MaC tool represents a requirement specification as an abstract syntax tree [Hs02a]. JTrek traverses the Java class files to be monitored. Instrumentation inserts new code. The inserted code can access the contents of various run-time data structures, such as the run-time and local variable stacks, and will, when eventually executed, emit events carrying the extracted information to the observer [atUoP, Hs01].

There are two steps to the instrumentation process:

1. First, the JTrek-based application is run to instrument the target program.
2. Next, the target program is executed to allow the instrumented code to generate the desired log of run-time activity.

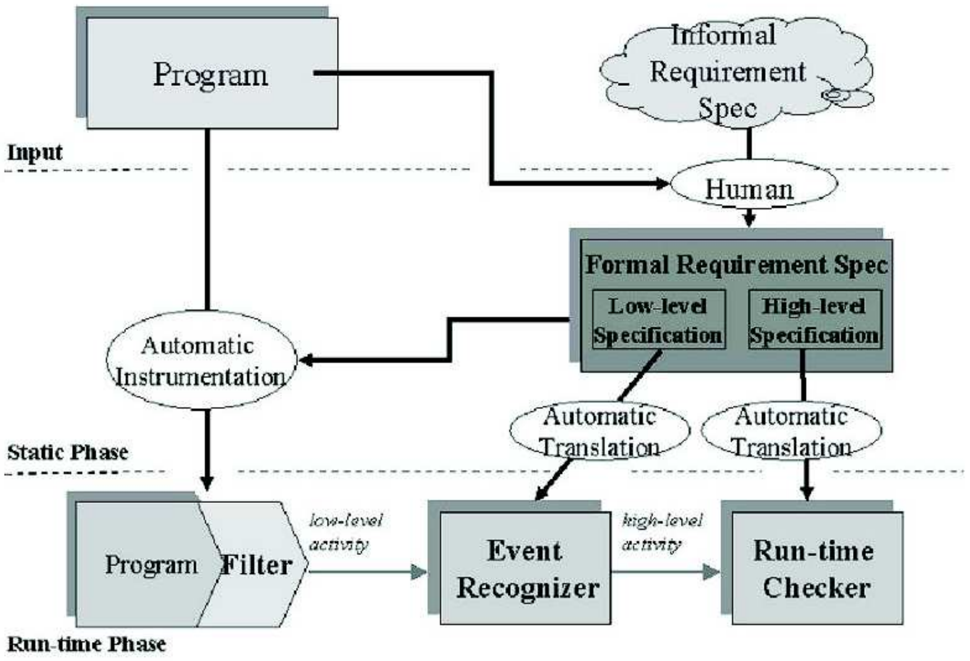

Figure 3.1: Overview of the MaC Architecture

3.3 Overview of the MaC Architecture

The structure of the architecture is shown in Figure 3.1. The architecture involves two main divisions: Static phase and Run-time phase. Given a target program and a formal requirement specification, the static phase (which occurs before a program is run) involves the automatic generation of run-time components including a *Filter*, an *Event Recognizer*, and a *Run-time Checker*. The run-time phase (which occurs during the execution of a target program), involves the collection of information of the target program execution and its check against a formal requirement specification [Kim99].

3.3.1 Static Phase of the MaC Architecture

Run-time components of MaC are generated from PEDL and MEDL specifications in the static phase [Kim99]. MaC has three static phase components: an *instrumentor*, a *PEDL compiler*, and a *MEDL compiler*.

A MaC instrumentor takes a Java bytecode (`*.class`) and instrumentation information (`instrumentation.out`) containing a list of monitored variables/methods and monitoring flags in the PEDL specification [Kim99]. Based on these two inputs, the MaC instrumentor inserts a monitor (Filter) into the target bytecode. A PEDL compiler compiles a PEDL specification into an abstract syntax tree (AST) (`pedl.out`), which is evaluated by an Event Recognizer at run-time. At the same time, a PEDL compiler generates instrumentation information (`instrumentation.out`) which is used by the instrumentor. Similarly, a MEDL compiler compiles a MEDL specification into an AST (`medl.out`), which is evaluated by a Run-time Checker at run-time.

Note that MaC uses JavaCC and JJTree to compile its PEDL and MEDL specifications. JavaCC is an environment for writing parsers and for generating and manipulating abstract syntax trees (AST) [Mic02]. JJTree is a preprocessor for JavaCC, that inserts parse tree building actions at various places in the JavaCC source. The output of JJTree is run through JavaCC to create the parser. JJTree provides access to the parse tree nodes from within grammar actions: push, pop and otherwise manipulate parse tree contents however appropriate.

Run-time components of the MaC architecture are generated automatically from a target program and a formal requirement specification [Kim99]. The MaC run-time components consist of:

- an instrumented target program, called *Filter*

- a low-level activity (PEDL) monitor, called *Event Recognizer*
- a high-level activity (MEDL) monitor, called *Run-time Checker*

Separation of the low-level requirement specification from the high-level requirement specification offers one significant advantage: This separation provides a description of high-level requirement specification by abstracting out the implementation specific details [Kim99]. A user need not know, interact with, or even worry about low-level details of the implementation language in order to write a specification for requirements of the system.

3.3.2 Run-time Phase of the MaC Architecture

The run-time phase of the MaC architecture consists of three components: a Filter, an Event Recognizer, and a Run-time Checker. At run-time, low-level information of the instrumented target program execution is extracted and reported to the Event Recognizer. Based on the values of the monitored entities it receives from the Filter, the Event Recognizer detects the occurrence of events according to a low-level specification written in PEDL, and sends them to the Run-time Checker. The Run-time Checker then checks the correctness of the target program execution according to a high-level requirement specification written in MEDL, based on the information of events it receives from the Event Recognizer [Kim99].

3.4 The MaC Languages

The MaC languages consist of Primitive Event Declaration Language (PEDL) and Meta-Event Declaration Language (MEDL). PEDL describes what to monitor in the

```
    <C> ::= c | defined( <C> ) | [ <E> , <E> )
            | ! <C> | <C> && <C> | <C> || <C>
            | <C> => <C>
    <E> ::= e | start( <C> ) | end( <C> )
            | <E> && <E> | <E> || <E>
            | <E> when <C>
```

Table 3.1: The Syntax of Conditions and Events

target program and is used to generate the Event Recognizer. MEDL describes the requirements that the target program must satisfy and is used to generate the Run-time Checker [Kim99].

3.4.1 Primitive Event Declaration Language (PEDL)

The design of PEDL is based on low-level (implementation-specific) details to describe what to monitor in the target program. Declarations of monitored entities are by necessity, specific to the implementation language of the target system. Furthermore, the name of the language reflects the fact that PEDL specifications are limited to primitive entities. PEDL is based on a logic for events and conditions similar to that of interval past-time Linear Temporal Logic (LTL). The models for this logic are sequences of states. Each state has a description of the truth values of primitive conditions and occurrences of primitive events. The syntax of conditions and events is given in Table 3.1 [Kim99, KKLS01].

The condition `defined(c)` is true whenever the condition `c` has a well-defined value, namely *true* or *false*. There are some events associated with conditions, namely the instant when the condition becomes *true* (`start(c)`), and the instant when the condition becomes *false* (`end(c)`). Also, any pair of events $\mathtt{e}_1$ and $\mathtt{e}_2$, define an interval of time and form a condition $[\mathtt{e}_1,\mathtt{e}_2)$ that is *true* from event $\mathtt{e}_1$ until event

e_2. Finally, the event (`e when c`) is present if `e` occurs at a time when condition `c` is *true* [KKLS01].

Defining Events

The primitive events in PEDL correspond to updates of monitored variables and invocations/returns of monitored methods [Kim99]. The event `update(x)` is triggered when variable `x` is assigned a value. The value associated with this event is the new value of `x`. Events `startM(f)` and `endM(f)` are triggered when control enters method `f` and returns from `f`, respectively. For example,

```
event OpenGate = startM(Control.open())
```

defines an event to be triggered as soon as a controller starts to open a gate.

All operations on events defined in the logic can be used to construct more complex events from these primitive events [Kim99]. Events have associated with them attribute values. These can be accessed using the attribute `value`.

Value

`value(e,i)` gives the $\mathtt{i}^{th}$ value in the tuple of values associated with primitive event `e`, provided `e` occurs [Kim99]. Values of primitive events are defined as follows:

- A value of `update(var)` is a tuple containing the current value of variable `var`.
- Values of method events `start(method)` and `end(method)` have not been implemented in the original version of MaC

3.4.2 Meta Event Declaration Language (MEDL)

The goal of a MEDL script is to provide a high-level requirement specification, by abstracting out implementation specific details. Like PEDL, MEDL is also based on logic for events and conditions: Primitive events and conditions in MEDL specifications are imported from PEDL specifications. More complex events and conditions are then built up using various connectives. The correctness of the system is described in terms of safety properties and alarms.

Past-time LTL is especially useful for safety properties. Havelund and Roşu explain,

> "These properties are very suitable for logic-based monitoring because they only refer to the past, and hence their value is always either true or false in any state along the trace" [Hs01].

The overall structure of a MEDL specification is given in Table 3.2 [Kim99, KKLS01].

A MEDL specification consists of five sections [Kim99]:

- the *import section* declares a list of events and conditions to be imported from an event recognizer.
- the *auxiliary variable declaration section* declares a list of auxiliary variables.
- the *event and condition definition section* defines events and conditions based on the imported events and conditions and auxiliary variables.
- the *property and violation definition section* defines safety properties and violations.

```
ReqSpec name_of_requirement_script

    /* Import section */
    import event e;       // imported event declarations
    import condition c;   // imported condition declarations

    /* Auxiliary variable declaration section */
    var int x;        // auxiliary variable declarations

    /* Event and condition definition section */
    event e2 = ...;        // event definition
    condition c2 = ...;    // condition definition

    /* Property and violation definition section */
    property c3 = ...;     // safety property definition
    alarm e3 = ...;        // violation definition

    /* Auxiliary variable update section */
    e -> { x‘ := ... ; }    // auxiliary variable update
End
```

Table 3.2: Structure of a MEDL Specification

- the *auxiliary variable update section* defines rules for how and when auxiliary variables are updated.

3.4.3 Example

In this subsection, we illustrate the use of PEDL and MEDL using a simple but representative example [Kim99]. The example is inspired by the railroad crossing problem, which is routinely used as an illustration of real-time formalisms. The system is composed of a gate, trains, and a controller. The gate opens and closes. The trains pass through the crossing. The controller is responsible for closing the gate when a train approaches the crossing and opening it after it was passed.

The following code shows a fragment of the gate controller implemented as a Java class.

```
public class Control {
    public static final int GATE_UP     = 0;
    public static final int GATE_DOWN   = 1;
    public static final int IN_TRANSIT  = 2;
    int gatePosition;
    public void open()  {...}
    public void close() {...}
        ...
}
```

In this example, we monitor the controller of the gate, using the requirement that the gate is down within 30 seconds after event `CloseGate` is sent, unless event `OpenGate` is sent before the time elapses.

Figure 3.2 shows the PEDL specification for the railroad crossing example. This PEDL specification declares two methods in the target program for monitoring: methods `open()` and `close()` of the `Control` class. In addition, two variables in the target program are declared for monitoring: integers `gatePosition` and `GATE_DOWN`

```
MonScr RailRoadCrossing

    export event OpenGate, CloseGate;
    export condition Gate_Down;

    monmeth void Control.open();
    monmeth void Control.close();
    monobj int Control.gatePosition;
    monobj int Control.GATE_DOWN;

    event OpenGate = startM(Control.open());
    event CloseGate = startM(Control.close());

    condition Gate_Down = (Control.gatePosition == Control.GATE_DOWN);
End
```

Figure 3.2: Excerpt of the PEDL Specification for the Gate Controller

of the `Control` class. Next, the PEDL specification introduces the high-level events `OpenGate` and `CloseGate`, as well as condition `Gate_Down`. Event `OpenGate` will be raised when the `open()` method of the `Control` class is invoked. Event `CloseGate` will be raised when the `close()` method of the `Control` class is invoked. Condition `Gate_Down` is true as long as monitored variable `gatePosition` of the `Control` class is equal to the monitored variable `GATE_DOWN` of the `Control` class.

The correctness requirement for the gate is given in Figure 3.3. The MEDL specification uses the events and condition imported from the PEDL specification. Next, two auxiliary variables are declared: `lastClose` and `currentTime`. A safety property `GateClosing` states that if there was a `CloseGate` event at the time when the gate was not down, which was not followed by either event `OpenGate` or condition `Gate_Down` becoming true, then the time allotted for gate closing has not elapsed yet.

```
ReqScr SafeCrossing

    import event OpenGate, CloseGate;
    import condition Gate_Down;

    var float lastClose;
    var float currentTime;

    property GateClosing =
        [ CloseGate when !Gate_Down,
          OpenGate || start(Gate_Down) ) => lastClose+30 > currentTime;

    CloseGate -> {
        lastClose‘ = time(CloseGate);
    }
End
```

Figure 3.3: Excerpt of the MEDL Specification for the Gate Controller

The time of the last occurrence of event `CloseGate` is recorded by auxiliary variable `lastClose`.

3.5 Summary

MaC offers a run-time framework that provides necessary assurance of software systems in a real-time environment [Kim99]. Formal monitoring and requirement specification languages (PEDL and MEDL) describe properties based on instantaneous events and continuous conditions. PEDL is based on low-level (implementation-specific) details to describe what to monitor in the target program. It is defined to describe primitive events in terms of program entities such as method calls and variables. The MaC architecture uses an approach that separates monitoring program-dependent, low-level activities from checking high-level activities. High-level activities, which are mapped from the low-level activities, are checked according to high-level requirement specification provided by MEDL.

Secondly, the MaC software is applied to executable code, not source code which is available to its developers only [LBAK$^+$99]. Instrumentation is performed directly on the executable code (bytecode, in the case of Java). Instrumentation is automatic, which is made possible by the low-level description in the monitoring script.

Furthermore, the components of the MaC architecture, Filter, EventRecognizer, and Run-time Checker, are automatically generated from the PEDL and MEDL specifications. These components do not require a specialized execution environment such as a special hardware or specific Operating System (OS)/Virtual Machine (VM) [Kim99].

3.6 Application of MaC to Monitoring and Control of Resource Consumption

We present work on an extension of MaC, to monitor aspects of resource consumption in a mobile code environment. MaC is a tool that performs run-time formal analysis to provide assurance on the correctness of software execution [KKLS01]. Run-time formal analysis takes a program and a formal requirement specification as inputs. Then, when a program is running, the execution of the program is checked by the formal requirement specification at run-time. Run-time formal analysis can find errors and help users to take a recovery action before critical failure happens to the system.

Original MaC, as described in this chapter, cannot be applied to the monitoring and control of system resource use. As a result, we have extended MaC to the enforcement of security policies governing ways to better monitor and control resource consumption by Java applets. Original MaC is restricted to communication of primitive events and cannot communicate, for instance, the parameters of monitored methods [KKLS01]. This is not sufficient if MaC is to monitor, for instance, file resource consumption, where filenames are expressed in String objects and file resource use is performed through access to methods in the Java core classes (API). In addition, in order to accurately monitor the resource utilization of Java applications, the application code as well as the libraries used by the application – in particular classes of the Java Development Kit (JDK) – have to be transformed.

In the following chapter *Extensions*, we describe several hurdles we have encountered in using MaC to monitor the resource use of Java applications and present solutions to these problems through extensions to MaC.

Chapter 4

Extensions

In this chapter, we describe our extensions to the MaC software to the enforcement of security policies. These extensions are aimed at providing ways to monitor and control resource consumption by Java applets. Specifically, our focus is on control of file system resource use.

Firstly, to restrict access of foreign pieces of code to file system resources, it is necessary to monitor application program interface (API) methods. Whenever an instruction tries to access a system resource, Java's API library is called [HK02]. It is our goal to design a mechanism that enables a user to customize his or her own version of personally designed security policies that place constraints on how an applet uses file system resources. As such, a methodology to monitor Java API methods is needed, since these methods control access and use of file system resources.

Next, with the ability to monitor Java API methods, it is necessary to extend MaC with the ability to monitor file system use. Using MaC to monitor file system use involves the following three steps:

1. **Allow Method Arguments to be Communicated**

   For security purposes, it is necessary to have access to the values of arguments to API methods that we wish to monitor. API methods perform actions on system resources [Yak02]. In order to place limits on resource use, arguments are necessary because they allow the resources to be identified.

   Kim, Viswanathan, Ben-Abdallah, Kannan, Lee, and Sokolsky explain,

   > "The user should be able to reason about any program object that holds a value and thus forms part of the program state, and any event that the program can engage in. The former includes individual variables and composite data structures (objects in Java). The latter, at least in Java programs, includes only method calls and returns (which also captures other activities, such as process creation and communication)" [KVBA$^+$98].

   In original MaC, each monitored variable event (`update(var)`) has an associated value: The value associated with this event is the value of that variable. However, method events (`startM(method)` and `endM(method)`) do not have a corresponding value associated with them [Kim99]. For security purposes, it is desirable for the value of method events associated with the start of the monitored methods, to be given by the values of all arguments to the method. In addition, original MaC is restricted to primitive values. As a result, a first step toward extending MaC to monitor file system use, is to provide the communication of primitive values of methods.

2. **Allow Strings to be Communicated**

   It is necessary to augment the types of values of method invocation events, with Strings. A String value is a non-primitive object that consists of a stream of characters. Communication of String values is necessary if communication is to handle filenames: Filenames identify files. Typically, filenames are expressed as Strings. Furthermore, many file resource API methods require filenames as arguments. It is necessary to communicate these arguments to keep track of the identity of files. This extension is reflected in changes to the PEDL and MEDL grammars given in Appendix A and B, respectively.

3. **Add Sets to MEDL**

   Some sort of a collection/set is needed as a storage mechanism to collect and keep track of the use of different files. Sets in MEDL provide a means of retaining file identities so that policy specification can restrict, specify, and limit file system use. In addition, in original MaC, like PEDL, MEDL is restricted to primitive types. This extension is reflected in changes to the MEDL grammar given in Appendix B.

With the facility to monitor API methods and the above three extensions to MaC, user access to files can be monitored, and MaC can uniquely identify files and monitor operations (open, read, write, close, etc.) on specific files. In short, aspects of file resource usage can be monitored. This chapter proceeds to describe each extension. Section 4.1 describes our technique to monitor API methods. Section 4.2 describes the three extensions necessary to MaC for it to monitor file system use.

4.1 Monitor API Methods

Java API methods provide a set of classes that the Java system uses to access system resources [HK02]. Therefore, it was necessary to monitor API methods to account for and restrict system resource consumption. In particular, our work looks at file system resource use.

However, MaC cannot monitor API methods directly: MaC cannot monitor methods invoked on an object within a class [LBAK$^+$99]. Secondly, API classes are tightly interwoven with native code of the Java run-time system, which causes subtle complications for the rewriting of Java API classes [HK02]. For these reasons, when a running program attempts to access a file system resource of which it is desirable to monitor resource consumption, in our approach we redirect program control to a subclass of the API class invoked. For example, if an applet tries to open a file, the applet will try to invoke a constructor method of class `java.io.FileInputStream`. The constructor method

```
java.io.FileInputStream.FileInputStream(String)
```

opens a connection to a file.

In our work, we would subclass
`java.io.FileInputStream` with class `FileInputStream`. We do not change Java API classes. Instead, we provide subclasses for API methods we wish to monitor as shown in Figure 4.1, that redirect their calls to subclasses, that merely serve to invoke the corresponding API methods themselves. With this scheme, the use of the API is also where it is defined. This fulfills the MaC requirement that the monitored object or method be in the class where it is defined.

Example 4.1.1 Monitor API Methods.

In this section, we provide an example of our API subclass `BufferedReader.java` given in Figure 4.1 and its monitoring specification given in PEDL in Figure 4.2.

Our approach to monitor calls to Java API methods is based on the class modification and method-level modification techniques introduced by Chander, Mitchell and Shin [CMS01]. The main techniques of our approach involve subclassing non-final API classes, and containing one object in another (composition) when access to final classes is desired.

Java uses the API class `BufferedReader.java` to handle input efficiently. That is, it wraps another reader, so that when large chunks of data are read in, `BufferedReader.java` can be used to retrieve data in smaller, more manageable bits [Mic03b]. Since our work was to monitor file resource access, a subclass for `BufferedReader.java` was created.

An excerpt of subclass `BufferedReader.java` is given in Figure 4.1. `BufferedReader.java` is a subclass that extends API class `BufferedReader.java`, but intercepts calls to methods `readLine()` and `close()` and merely serves to call the corresponding API methods. API method `BufferedReader.readLine()` performs the task of reading text; API method `BufferedReader.close()` closes the input/output (I/O) stream.

It was necessary to create a subclass for each API class that we wished to monitor. The subclasses insure that we could properly specify API use in the MaC language PEDL. Declarations of monitored entities takes place in the PEDL script.

An excerpt of the PEDL specification is given in Figure 4.2. The PEDL specification associates events with the invocation and termination of monitored methods.

```
public class BufferedReader extends java.io.BufferedReader {

    public BufferedReader(java.io.Reader in) {
        super(in);
    }

    public String readLine() throws java.io.IOException {
        return super.readLine();
    }

    public void close() throws java.io.IOException {
        super.close();
    }

}
```

Figure 4.1: Excerpt of Subclass `BufferedReader.java`

Three methods are monitored: `permissions.run()`, `FileInputStream.FileInputStream(String)`, and `BufferedReader.close()`. Next, three events are defined and exported: `initialize, fileOpened,` and `fileClosed`. Note that specification of `monmeth FileInputStream.FileInputStream(String)` and `monmeth BufferedReader.close()` refers to subclass `FileInputStream.java` and `BufferedReader.java`, where the API methods `FileInputStream(String)` and `close()` are used, respectively.

Implementation 4.1.2: Monitor API Methods

This section describes the implementation of our approach to monitoring Java API methods. Our approach to monitor calls to Java API methods is based on the

```
MonScr permissions
    export event initialize, fileOpened, fileClosed;

    monmeth void permissions.run();
    monmeth void FileInputStream.FileInputStream(String);
    monmeth void BufferedReader.close();

    event initialize = startM(permissions.run());
    event fileOpened =
        startM(FileInputStream.FileInputStream(String));
    event fileClosed = endM(BufferedReader.close());
end
```

Figure 4.2: Excerpt of PEDL Specification for Subclass `FileInputStream.java` and `BufferedReader.java`

class modification and method-level modification techniques introduced by Chander, Mitchell and Shin [CMS01]. The main techniques of our methodology involve subclassing non-final API classes, and containing one object in another (composition) when access to final classes is desired. In the following, we will describe a general methodology for creating the structure necessary for monitoring API methods.

Methodology for Creating an API Subclass

1. Call subclass' name the same name as its corresponding Java API class. Place subclass in the directory that the target application is in.

   It is not a compile-time error to give a class of the application system a name that is identical to a Java API class [GJSB00]. For example, the I/O import `import java.io.*;` causes the names of all public types declared in the package `java.io` to be available within the class and interface declarations of the compilation unit (application package). However, if an imported class conflicts with a class of the application package, the imported class is ignored in such cases.

2. Add `extend java.io.filename` in classname declaration

   For example:

```
public class BufferedReader extends java.io.BufferedReader {

}
```

By extending the Java API class, we are able to override the methods we wish to monitor. Furthermore, we can keep the original functionality of methods we do not wish to monitor, by letting the API extension take program control to where it is defined.

However, this subclassing technique cannot be done for final classes. Our approach to dealing with final classes is based on the composition technique introduced by Wells [Wel03]. Composition simply refers to containing one object within another.

(a) Create a class object in the global variable declaration section of the class through which you want to intercept an API call, so that this class will remember its process object each time it is called.

   For example:

```
public class Class {
    private java.lang.Class cl;

}
```

(b) Include all methods from the original API class `packagename.filename` that are to be monitored, and simply call each API method on the class object.

(c) Initialize the class object with, for instance, `cl = new java.lang.Class()` with necessary parameters given in the constructor method parameters as the first call in the composition constructor.

For example:

```
public class Class {
    private java.lang.Class cl;

    public Class () {
        cl = new java.lang.Class();
    }
}
```

This insures that the class object `cl` is initialized when the class is first instantiated. The class is first instantiated when the constructor method is called [Mic03a]. In addition, this insures that methods we want to monitor will call each API method on this initialized class object `cl`.

3. For any method whose declaration indicates that it throws an exception, include `java.io.ExceptionName` in its declaration.

   This insures that the appropriate API Exception class is called.

4. Declare all constructors that are called in the monitored Java code.

   This insures that all constructors called can be resolved within its corresponding subclass. By default, constructors are not inherited by subclasses.

5. Only include methods of the API that are to be monitored. Methods that are not monitored are inherited from the corresponding superclass in the API.

This insures that subclass construction is kept as simple as possible.

6. For constructor methods:

   First line of the constructor must call `super(...)` .

   When you instantiate a subclass, execution will begin with an invocation of the constructor in the base class [Mic03a]. Note that the argument list of `super(...)` must be identical to the formal parameter list of the constructor.

7. For non-constructor methods that return something:

   Include: `return super.methodName(...);` Same remarks about argument list expressed in 6., apply here.

   For instance,

   ```
   public String readLine() throws java.io.IOException {
       return super.readLine();
   }
   ```

   This calls the extended class' `readLine()` method.

8. For non-constructor methods that do not return something

   Include: `super.methodName(...)`; Same remarks about argument list expressed in 6., apply here.

   For instance,

   ```
   public void close() throws java.io.IOException {
       super.close();
   }
   ```

   This calls the extended class' `close()` method.

Limitations 4.1.3: Monitor API Methods

The use of subclasses to monitor API methods provides a way to monitor actions on system resources, but is not without some limitations.

Firstly, current subclass construction is performed manually. Section 4.1.2 outlines a methodology for creating subclasses appropriately. The subclass serves to redirect program call execution to a class that simply calls the corresponding API method, and thus provides a means to monitor those calls. However, the subclass must be constructed manually. Although an automatic design appears straightforward, this issue was designated for future work.

Another limitations of our technique to monitor API methods lies in its dependency on the Java language. We chose to use Java for two reasons: Firstly, the current implementation of MaC works on applications written in Java, despite the framework is generic and can apply to any language [KKLS01]. Secondly, Java easily tops the list of preferred platforms for Internet-savvy mobile code [MF99]. Most popular and used Internet browsers on the market, Netscape Navigator and Internet Explorer, are automatically configured to dynamically download and run Java code. However, Java is but one programming language for mobile code among many. The methodology we provided for constructing a subclass serves to provide a means to monitor Java API methods that make system resource calls.

4.2 Monitor File System Use

In this section, we describe our technique to monitor file system use. The current Java security model provides a permissions framework that specifies the host resources that mobile code may access [CMS01]. The problem is that the framework does not monitor the *extent* to which a resource is used. This means that code which may have legitimate use for a certain resource, may abuse this privilege. For example, the system may be rendered useless by greedy actions that monopolize CPU time, use all available system memory, or starve other threads and system processes. A mechanism that controls resource use is needed. In this work, we develop a mechanism that monitors and controls Java file system resource use.

Original MaC cannot support the monitoring of system resource use. Three essential limitations exist and had to be addressed in this research project:

1. The arguments of methods are not communicated.

   In order to track system resource use, it was necessary to enhance MaC's primitive language to communicate not only the value of monitored variables, but also communicate the arguments of monitored methods. Java API methods provide a set of classes that Java uses to access system resources. For tracking Java's use of system resources, the value of corresponding API monitored methods should be associated with the value of method inputs.

2. Both MaC's PEDL and MEDL languages do not recognize Strings.

   The MaC language must include Strings if MaC is to handle filenames, because filenames are typically represented as Strings. Since the goal of this research is to provide a mechanism that supports the monitoring and control of file resource use, a language that supports filename expression is necessary.

3. The MaC language has no means to keep track of the use of different files.

   The ability to handle some sort of a collection in MaC is essential to the monitoring and control of file resource use: It enables MaC to keep track of the resources that the applet has accessed during its execution.

Given the above limitations, three extensions to MaC were necessary to give it the ability to monitor file resource use:

1. Allow method arguments to be communicated
2. Allow Strings to be communicated
3. Add sets to MEDL

The above three extensions are discussed in the following three subsections. With these three extensions, operations (such as open, read, write, and close) on files can be monitored, and the identity of the files involved can be tracked.

4.2.1 Allow Method Arguments to be Communicated

We established communication of the value(s) associated with the start of monitored methods. The value(s) associated with the start of monitored methods are given by the values of all arguments. However, original MaC communication could not get at these method arguments.

Events `startM(f)` and `endM(f)` are triggered when control enters and returns from method `f`, respectively. For example,

```
event intToInt = startM(Integer.Integer(int));
```

expresses that the event `intToInt` should be raised when the constructor `Integer(int)` is invoked. The value of `intToInt` is the integer that was passed into the constructor.

```
alarm intToIntAlarm = intToInt;

intToInt -> {
  print "IN PERM.MEDL: intToIntAlarm";
  print "IN PERM.MEDL: value(intToInt,0)= " + value(intToInt,0);

}
```

Figure 4.3: Excerpt of MEDL Script for Example Allow Method Arguments to be Communicated.

Since methods may have zero, one, or many arguments, the events have a tuple of values associated with them. Individual elements can then be obtained via a projection, i.e., `value(e,i)` gives the `i`th value in the tuple of values associated with event `e`, provided `e` occurs. We define the value of an event `e` defined by event `e = startM(m)` as the tuple containing the arguments of method `m`, when `m` starts.

Example 4.2.1.1: Allow Method Arguments to be Communicated

An excerpt of the MEDL script for example Allow Method Arguments to be Communicated is given in Figure 4.3. The MEDL script raises an alarm when the event `intToInt` is raised. In addition, the MEDL script prints the value of the argument of the monitored method.

Implementation 4.2.1.2: Allow Method Arguments to be Communicated

The MaC architecture consists of three components: Filter, Event Recognizer, and Run-time Checker [KVBA+98]. The Filter extracts low-level information such as variable values, from the running code. In order to achieve this, MaC instruments the code of the system to be monitored. The Filter sends this information to the Event Recognizer. The Event Recognizer receives information from the Filter, recognizes events by evaluating the PEDL expressions that define them, and sends them to the Run-time Checker. The Run-time Checker will receive the events and check them against requirement specifications expressed in the MEDL script.

There are two parts to the implementation of the value tuple for `startM`: one consists of ensuring that the right values are passed from the Filter to the Event Recognizer. The second part consists of properly packaging these values in the vector. The MaC software provides a set of classes for invoking the target program, the Event Recognizer, and the Run-time Checker. For instance, the Filter thread is initialized through a method `init()`. The `init()` method of the Filter thread is instrumented into the main method of the application class [Kim99]. Then, the instrumentor adds an invocation of `init()` to the main method of the monitored code. Code to send a message to the Event Recognizer is also added to all other monitored class files. The code consists of calls to methods in `SendMethod.java`. If the event was defined through a method event, then the message contains the name of the called method, and its argument values in the case of `startM`. If the event was defined through a variable update event, then the message contains the name of the updated variable and its new value.

Firstly, we needed to address instrumentation: MaC performs instrumentation through invocation of its class `Instrumentor.java`. Recall from Chapter 3 that a PEDL compiler generates instrumentation information (`instrumentation.out`) which is used by the instrumentor. Ability to manipulate parse tree contents, which was gained through use of the JJTree preprocessor run through JavaCC to create the grammar parser, was used to retrieve method arguments and communicate them from Filter, to Event Recognizer, to Run-time Checker. The following sequence of instructions were added to `Instrumentor.java`:

1. The first instrumentation creates a call to a new method in `SendMethod.java`.
2. The second instrumentation passes a code so that the Event Recognizer knows that an event for some execution point is coming. Execution points refer to method invocation, method return, start and end of program, and method exception [LBAK$^+$99].
3. The third instrumentation passes the argument from the top of the local variable stack as the value of an event to `SendMethod.java`.
4. The fourth instrumentation passes the ID of the primitive variable to be communicated.

Java has two stacks: local variable stack and operand stack [MF99]. Method arguments in Java sit on the stack frame with local variables. In step 3, it was imperative to pass an argument from the top of the local variable stack, to insure we pass the value of a method argument.

Over in MaC class `SendMethod.java`, an additional method needed to be created, that took in the stack parameter value passed. This parameter was then sent to the Event Recognizer. In original MaC, after a connection between the Filter and the Event Recognizer is established, the Filter sends snapshots to the Event Recognizer. A snapshot consists of a ID field for a monitored variable and a value field of the

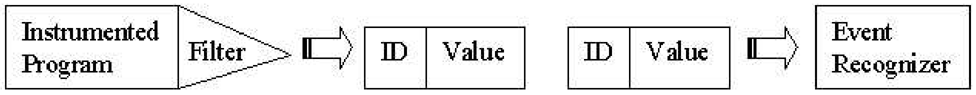

Figure 4.4: Example of Snapshots from Filter to Event Recognizer in Original MaC

monitored variable. A snapshot reporting a beginning/ending of a method has a boolean value field: *false* means beginning of a method and *true* means ending of a method. An example of snapshots from a Filter to an Event Recognizer in original MaC is illustrated in Figure 4.4.

So, in original MaC, execution points were passed by means of method

`SendMethod.sendMethod(boolean,int).`

The boolean argument reports a beginning/ending of a method and the integer argument contains the ID of the monitored object or method. For an event, ID is eventID, where eventID can be any number between -128 to 127. The Event Recognizer sends an object containing a table mapping eventIDs to event names. The Run-time Checker uses the table to recognize what an ID from the Event Recognizer means.

Over in MaC class `SendMethod.java`, an additional method needed to be created, that took in the stack parameter value passed. This parameter was then sent to the Event Recognizer.

However, in our extension we wanted to implement a value for `startM`. As a result, two new fields were added to the snapshot sent from Filter to Event Recognizer: a type-event-valueID to indicate the type of method argument passed (String, int, etc.), followed by the method argument value. An example of snapshots from a Filter to

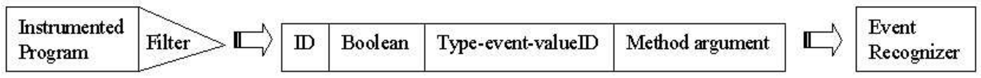

Figure 4.5: Example of Snapshots from Filter to Event Recognizer After Our Extension Allow Method Arguments to be Communicated

an Event Recognizer after this extension is illustrated in Figure 4.5.

In addition, a new method in MaC class `SendMethod.java` had to be defined:

`sendMethod(boolean,int,int)`.

The boolean argument reports a beginning/ending of a method, the second argument `int` contains the method argument, and the third argument `int` contains the ID of the monitored object.

The above changes to MaC classes `Instrumentor.java` and `SendMethod.java` ensured that the right value (the method parameter) was passed from the Filter to the Event Recognizer. The next step required changes to the Event Recognizer, to ensure it read off the incoming method arguments after receipt of a `startM` event. This step ensured that the method event values were properly packaged into a value tuple for the monitored method and sent to the Run-time Checker.

Limitations 4.2.1.3: Allow Method Arguments to be Communicated

Although communication of primitive values of method arguments is now possible with the extension “Allow Method Arguments to be Communicated”, MaC is still restricted to primitive values: That is, variables in PEDL and MEDL scripts can only range over the types `boolean, byte, char, short, int, long, float, double`. A restriction to primitive values limits the extent of system resource control, including that of file system resources. For example, filenames are typically represented by String type, a type outside the scope of Java primitive types. In this research project, we wanted to enable the ability to place limits on file system resource use. As a result, MaC's restriction to primitive constructs needed to be resolved. This limitation is addressed in our next extension.

Another limitation is that events are assigned at most one value. Methods take in zero, one, or *several* input parameters. Only the first method parameter, if it exists, makes up the value of a method invocation event. Although each method parameter is placed into a value vector, the communication of the entire argument list and not just the first argument is not yet implemented.

A PEDL compiler compiles a PEDL script into an abstract syntax tree. The abstract syntax tree consists of monitored method parameter values and updated variable values. The Event Recognizer evaluates the events and conditions defined in the PEDL abstract syntax tree when it receives a snapshot from the Filter. To overcome the limitation above, the abstract syntax tree has to be traversed to retrieve all of the parameter values sequentially stored on the local variable stack for each `startM` monitored method event. Each of these values would then have to be placed into their corresponding value vector.

4.2.2 Allow Strings to be Communicated

In this extension, we modified MaC to allow String communication between Filter, Event Recognizer, and Run-time Checker. Refer to Appendix A for changes to the PEDL grammar and to Appendix B for changes to the MEDL grammar. Original MaC communication was restricted to primitive definition.

The rationale for restricting MaC to monitoring only primitive variables was the following: If the size of an acceptable variable is not defined, MaC must always watch an arbitrarily large object graph [KVBA$^+$98]. This practice poses an unacceptable performance penalty. There is a tradeoff between the expressiveness of event definitions and the run-time cost of event detection. The language can be designed to allow very expressive event definitions so that violation of any requirement property is itself an event. However, the drawback of an expressive event definition language is the cost of event detection at run-time. In general, as the language becomes more expressive, the granularity of detection needs to be finer, which incurs more overhead in terms of time and space.

On the other hand, if the language has limited expressiveness, it cannot define some useful events. The current Java security model does not monitor the extent of use of host resources that foreign mobile code may access. A mechanism that controls system resource use is needed. With the goal of preventing resource-consuming attacks, any gain of ability to control resource use is worth a trade-off to performance.

In addition, extending MaC beyond primitive communication to a language that handles the communication of Strings, is not cause for severe performance degradation. In this extension, MaC's PEDL and MEDL grammars were extended to recognize Strings. Now, not only is String communication in MaC possible, but also the method arguments that are Strings can be communicated and referenced in the requirement specification.

```
monmeth void FileInputStream.FileInputStream(String);
event fileOpened =
    startM(FileInputStream.FileInputStream(String));
```

Figure 4.6: Excerpt of PEDL Script for Example Allow Strings to be Communicated.

```
alarm openedFileAlarm = fileOpened;

fileOpened -> {
  print "IN PERM.MEDL: value(fileOpened,0)= " + value(fileOpened,0);
}
```

Figure 4.7: Excerpt of MEDL Script for Example Allow Strings to be Communicated.

Example 4.2.2.1: Allow Strings to be Communicated

In original MaC, PEDL could only monitor changes to variables of primitive type. With this extension, now PEDL can also define an event related to Strings. With String communication enabled, MaC can be used to monitor updates to String variables and invocations of methods with String arguments.

An excerpt of PEDL script for example Allow Strings to be Communicated is given in Figure 4.6. The PEDL script monitors method

`FileInputStream.FileInputStream(String)` .

Also, this excerpt of our PEDL script specifies a method invocation event `fileOpened` whose values is the name of the file opened.

Figure 4.7 shows a MEDL script that raises an alarm whenever the `fileOpened` event is received and prints the name of the file opened.

Implementation 4.2.2.2: Allow Strings to be Communicated

Both MaC languages PEDL and MEDL needed to be extended to include Strings. Refer to Appendix A and B for extensions to the PEDL and MEDL grammars, respectively.

Recall from Chapter 3 that a PEDL compiler compiles a PEDL specification into an abstract syntax tree (AST) (`pedl.out`), which is evaluated by an Event Recognizer at run-time. Similarly, a Run-time Checker evaluates events and conditions defined in a MEDL AST (medl.out) when it receives an event or a condition from an Event Recognizer. Both PEDL and MEDL grammars needed to be addressed in this extension to allow Strings to be communicated:

1. The input to the PEDL code generator, a specification given in a separate file from the MaC application program, was modified.

2. The JJTree preprocessor for JavaCC was run to provide access to the parse tree nodes.

3. The JavaCC parser generator was run. Output from this parser generation would be read into the MaC application program at run-time.

Steps 1-3 were completed for the MEDL code generator as well.

Once PEDL and MEDL grammars were extended to include String definition, communication between the Filter and Event Recognizer had to be addressed. The Filter sends snapshots to the Event Recognizer: A snapshot consists of an ID field for a monitored variable and a value field of the monitored variable. The ID field for a monitored variable identifies if the entity represents an execution point, a global variable, or a local variable. A snapshot reporting the invocation or termination of a method has a boolean value field: *false* means the method has been invoked and *true* means the end of a method. Communication between Filter and Event Recognizer is read off byte-by-byte, since original MaC communication is restricted to primitive types. However, a String type represents a stream of bytes, so a data type identifier byte needed to be added to the stream. A data type identifier byte would serve to tell the stream reader how to interpret the following sequence of characters.

Changes to both MaC classes `Instrumentor.java` and `SendMethod.java` were made: Class `Instrumentor.java` was changed to pop method arguments of monitored methods from the top of the local variable stack. In addition, a new method in class `SendMethod.java` was defined. In original MaC, execution points `startM` and `endM` would invoke method `sendMethod(boolean, int)`. The boolean argument reports a beginning/ending of a method, and the integer contains the ID of the monitored object. Now, with an additional method argument (a filename) passed, a new method

```
sendMethod(boolean,String,int)
```

was defined. The boolean argument reports a beginning/ending of a method, the second argument `String` contains the String value of the method argument, and the third argument `int` contains the ID of the monitored object. An example of snapshots from Filter to Event Recognizer after this extension is illustrated in Figure 4.8

Furthermore, changes needed to be made to the Event Recognizer and Run-time Checker: Upon recognizing a method invocation event, a data type identifier byte was placed on the stream that links communication of the Event Recognizer to the Run-time Checker. At the Run-time Checker end, upon recognizing a method invocation event, the checker would first read off the data type identifier byte and then interpret

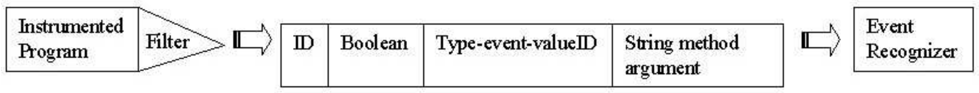

Figure 4.8: Example of Snapshots from Filter to Event After Our Extension Allow Strings to be Communicated

the following character appropriately based on the value of its preceding identifier byte.

Limitations 4.2.2.3: Allow Strings to be Communicated

With MaC communication able to handle Strings, MaC is no longer restricted to primitive values. However, String is but one of several non-primitive types. Non-primitive types in Java include user-defined classes and collections such as arrays and vectors.

In the case of policy definition of file system resources, it is relevant to be able to handle not only filenames, but file descriptors as well. A file descriptor is a handle created by a process when a file is opened [Mic03a]. Without the ability to communicate more complex objects such as file descriptors, a policy that limits currently opened files cannot be specified: This is because the closing of a file does not take in any arguments (such as a filename), but rather is performed on its file descriptor. Consequently, even with our extensions, it is still impossible to specify and determine the identity of a file involved in a close operation on the Run-time Checker side.

On another note, MaC uses Data Streaming communication over a Pipe. Communication via Data Streaming allows threads to send events and receive primitive and String values via an underlying input stream [Mic03a]. However, a design that would allow primitive data types *and* all object types requires communication via Object Streaming. Although in this research project implementation of an Object Streaming over Pipe approach was begun, efforts were diverted to extending MaC's current communication via Data Streaming. Two main reasons explain this choice: Firstly, rather than introduce the potential for errors by replacing MaC's original approach to communication, extension of its existing communication mechanism was a more practical idea. Also, Object Streaming over Pipes does not appear to be particularly reliable in the current version of Java. Documented problems include: exceptions thrown for reasons not understood, reader or writer threads held in infinite loops, and premature termination of threads. The communication of arbitrary objects between the threads in MaC would thus not only require MaC's communication to be based on Object Streaming over Pipes, but also a more stable release of Java.

4.2.3 Add Sets to MEDL

With the ability to use MaC to retrieve the arguments of Java API file system resource calls, it is possible to communicate them to MaC's Run-time Checker. For example, if a call is made to Java's API method `FileInputStream.FileInputStream()` to open a file, the value of the corresponding method invocation event is now assigned the name of the file. Our next extension will allow the Run-time Checker to collect these filenames and store them in a set. To this end, it was necessary to extend MEDL with a Set data type. Refer to Appendix B for the addition of sets in MEDL. Facility of sets in MEDL allow for the values of events to be collected. This, in turn, allows a number of policies relevant to the control of file resource use to be specified. For example, it is now possible to limit the input/output (I/O) activity of a Java

```
var String valueFileOpened;
var Set filesOpenedSet;

alarm openedFileAlarm = fileOpened;
alarm fileClosedAlarm = fileClosed;

initialize -> {
  filesOpenedSet’ = emptySet;
}

fileOpened -> {
  valueFileOpened’ = value(fileOpened,0);
  filesOpenedSet’ = add(valueFileOpened,filesOpenedSet);
  print "IN PERM.MEDL: filesOpenedSet= " + filesOpenedSet;
}
```

Figure 4.9: Excerpt of MEDL Script for Example Add Sets to MEDL.

applet: limits can be placed on the total number of files an applet tries to access. The filename can be checked for membership in a set collecting the different files an applet tries to open. As soon as the size of that set exceeds a certain limit, an alarm is raised.

Example 4.2.3.1: Add Sets to MEDL

An excerpt of the MEDL script for example Add Sets to MEDL is given in Figure 4.9. The MEDL script initializes set `filesOpenedSet` to be empty. Next, whenever a file is opened, event `fileOpened` is raised and the name of the opened file is added to set `filesOpenedSet`.

With the facility to monitor API methods and the above three extensions realized, many interesting policies can now be specified in MaC to control resource use. The next chapter will illustrate them through several examples.

Implementation 4.2.3.2: Add Sets to MEDL

The addition of sets in MEDL involved the addition of sets into the MEDL grammar. Refer to Appendix B for addition of sets into the MEDL grammar. Furthermore, the String data type was added to the MEDL grammar so that Strings could be expressed in requirement specification. Next, the MEDL grammar had to be parsed. The Java Compiler Compiler (JavaCC) is the parser generator MaC uses to read the MEDL grammar specification and generate a Java program that can parse MEDL scripts.

In addition to changes to the MEDL grammar, changes to a symbol table needed to be made. A symbol table is derived from a set of Java class files provided as part of the MaC software. There are three types of nodes in the abstract syntax tree of a MEDL script: condition node, event node, and expression node. A node referring to a program variable (a method) has a link to an entry in the MaC symbol table, which contains both type and value of monitored variables. A set was added to the MaC symbol table definition, for persistent storage of its contents. To simplify the implementation, a set in MEDL was implemented in terms of `java.util.Set`.

In addition, operations `add, emptySet, size,` and `print` were added to the functionality of sets in MEDL.

- `set = add(x,set)` Returns a set with the specified element added to this set if it is not already present.
- `emptySet` The set with no elements.
- `set = size(set)` Returns the number of elements in this set (its cardinality).
- `print(set)` Prints the contents of the set.

The operations implemented for the functionality of sets in MEDL offered a good selection of resource-aware policies that limit files resource use to be expressed. For example, with set operation `add` policies that need to collect and retain file identities can be expressed. Set operation `emptySet` enable set identity tests. Set operation `size` provides set cardinality so that limits can be placed on the amount of file access. Lastly, set operation `print` lets the contents of the set to be printed.

Limitations 4.2.3.3: Add Sets to MEDL

The inclusion of sets in MEDL is not without some limitations. Firstly, the list of operations on sets implemented in MEDL is not complete. Common set operations include `add, contains, equals, isEmpty, remove,` and `size`. Our implementations of set operations in MaC includes set operations `add, isEmpty,` and `size`. However, set operations `contains, equals,` and `remove` were not implemented. As a result, policies that require set membership tests, set equality, and set subtraction cannot be expressed. For instance, policies that require collecting the names of *currently* open files cannot be expressed. On the other hand, a good selection of policies were already possible with the addition of set operations `add, isEmpty,` and `size`. Secondly, our implementation of sets in MEDL currently limits MaC to the use of one set.

With the extensions to MaC discussed in this chapter, MaC can now be used to the enforcement of certain kinds of resource-aware policies. A good selection of policies are possible. In the next chapter, a number of example policies that limit file resource use are described.

Chapter 5

Examples

5.1 Small Examples

We demonstrate five MaC examples of resource-aware policies in this section. These policies show how we made use of the syntax of MaC languages PEDL and MEDL to integrate file resource control by means of the extensions described in the previous chapter.

In our first example, we present a policy that limits the read and write operations on a particular file. Our policy specifies that an alarm will sound for any attempt to read from a file and write to a file.

Our second example shows how to print the names of files which have had actions exercised upon them. The ability to print the filenames serves to keep a log of input/output activity.

In our third example, a policy that limits the total number of files the user has tried to access, is described. This example involves the ability to count the total number of files, a capability that was added through our work.

Our fourth example presents an example policy that restricts access to a particular file that the user might not want a segment of foreign code to access.

Given the extensions this research project has brought and the language operators in MaC (such as disjunction, conjunction, implication, equivalence, etc.), numerous varied policies can be expressed which monitor and control file resource consumption. Our fifth example illustrates this expansive reach: In our fifth example, we illustrate an example policy that specifies that *certain* files can only be accessed a certain number of times.

It is important to point out that none of these policies were expressible or enforceable in the original version of MaC. Moreover, our list of example policies is not exhaustive. The examples provided in this chapter serve to illustrate how the extensions presented in Chapter 4 can be used to develop a number of interesting resource-aware policies relevant to the security of file resource use. In summary, this thesis has extended the MaC software to allow the enforcement of certain kinds of security policies that monitor and control resource consumption by Java applets.

5.1.1 Example: Limit Read/Write

In this first example, we present a policy that limits the read and write operations of a segment of code.

Original MaC had no way to monitor system resource usage, communicate values of methods used to provide access to system resources, and understand String types which is necessary to communicate filenames. The extension that we need for this example is the ability to monitor API methods.

The design of this example's PEDL script in Figure 5.1 proceeds as follows: Two

```
monmeth String BufferedReader.readLine();
monmeth void PrintWriter.println();

event fileRead = startM(BufferedReader.readLine());
event fileWrite = startM(PrintWriter.println());
```

Figure 5.1: Excerpt of PEDL Script for Example Limit Read/Write.

```
alarm fileReadAlarm = fileRead;
alarm fileWriteAlarm = fileWrite;
```

Figure 5.2: Excerpt of MEDL Script for Example Limit Read/Write.

API methods are monitored: `BufferedReader.readLine()` which is used to read text from a file, and `PrintWriter.println()` which is used to write text to a file. Next, invocation of these methods are mapped to events. Event `startM(f)` is triggered when control enters method `f`.

The security requirements that need to be enforced are written in MaC language MEDL. The requirement uses the events imported from the PEDL script: An alarm is raised whenever a segment of foreign code tries to read from a file. Code uses method `BufferedReader.readLine()` to read from a file. Similarly, an alarm is raised whenever a segment of foreign code tries to write to a file. Code uses method `PrintWriter.println()` to write to a file. An excerpt of the MEDL script for Example Limit Read/Write is given in Figure 5.2.

This policy can also be applied to specify that a read operation be allowed from, but not allowed a write operation to, a particular file. Using the logic for events and conditions described in Appendix A, complexity can be added using the Java boolean connectives included in the language.

5.1.2 Example: Print Filenames

In this second example, we show how to print the names of files which have had actions exercised upon them. The ability to print the filenames serves to keep a log of input/output activity.

An auxiliary variable `filesOpenedSet` initializes a set. Auxiliary variables can be used to record certain aspects of the current execution. To be able to print the filenames, we needed to define a variable type that could collect them. In addition we needed to extend the print function of original MaC, to record each element of the collection. In this example, an auxiliary variable of set type was added, that would collect the names of files which had actions exercised upon them.

In keeping with the standard operations on sets, two necessary operations were added to MaC's language (as described in the previous chapter): `add(x,set)` and `emptySet`. `add(x,set)` adds the specified element `x` to the set `set`. The expression `emptySet` evaluates to the empty set.

An excerpt of the MEDL script for example Print Filenames is given in Figure 5.3. The MEDL script initializes a set `filesOpenedSet` to be empty. Event `initialize` is specified so that the set (and other variables, if present) can be initialized. This event was defined in PEDL with `startM(f)`, to assure that when control is first passed to the thread, that event `initialize` is triggered. Next, whenever the `fileOpened` event is received, the name of the file is added to the set `filesOpenedSet` and the contents of the set are printed.

```
  var String valueFileOpened;
  var Set filesOpennedSet;

  initialize -> {
    filesOpenedSet’ = emptySet;
  }

  fileOpened -> {
    valueFileOpened’ = value(fileOpened,0);
    filesOpenedSet’ = add(valueFileOpened,filesOpenedSet);
    print "IN PERM.MEDL: filesOpennedSet= " + filesOpenedSet;
  }

End
```

Figure 5.3: Excerpt of MEDL Script for Example Print Filenames.

5.1.3 Example: Limit Amount of File Access

This third example specifies a policy that limits the total number of files the user has tried to access. An alarm `exceeedingAllowableSize` is raised when the number of files the user tries to access exceeds a given number.

To establish a count on the number of elements in a set, a cardinality operation was needed: Operation `size(set)` was added to MEDL. Operation `size(set)` returns the number of elements in the set.

An excerpt of the MEDL script for example Limit Amount of File Access is given in Figure 5.4. The MEDL script initializes a set `filesOpenedSet` to be empty. Next, whenever a file is opened, the event is captured by event `fileOpened` and the set `filesOpenedSet` is added to with element `valueFileOpened`. Note that `size(filesOpenedSet)` returns the number of *different* files that have been opened.

```
var String valueFileOpened;
var Set filesOpenedSet;

alarm exceedingAllowableSize = start( ( size(filesOpenedSet) ) >= 200 )

initialize -> {
  filesOpenedSet' = emptySet;
}

fileOpened -> {
  valueFileOpened' = value(fileOpened,0);
  filesOpenedSet' = add(valueFileOpened,filesOpenedSet);
}
```

Figure 5.4: Excerpt of MEDL Script for Example Limit Amount of File Access.

Original MaC allowed counting the number of times an event has been raised. Operation `size(filesOpenedSet)` allows for `fileOpened` events to be counted, but not the number of different files.

5.1.4 Example: Restrict Access to Particular File

In this fifth example, we present a policy that restricts access to a specified file that the user might not want a segment of foreign code to access. Suppose the user wants to restrict access to a file named "passwd". To monitor and enforce this requirement, the names of files opened must be communicated from the Event Recognizer to the Run-time Checker.

As a result, method `FileInputStream.FileInputStream(String)` is specified to be monitored the excerpt of PEDL Script for Example Restrict Access to Particular File given in Figure 5.5. The method argument type `String` will be the name of

```
MonScr permissions
    export event initialize, fileOpened;

    monmeth void permissions.run();
    monmeth void FileInputStream.FileInputStream(String);

    event initialize = startM(permissions.run());
    event fileOpened = startM(FileInputStream.FileInputStream(String));
End
```

Figure 5.5: Excerpt of PEDL Script for Example Restrict Access to Particular File.

```
  alarm exceedingAllowableValue = start((value(fileOpened,0))==‘‘passwd’’);
```

Figure 5.6: Excerpt of MEDL Script for Example Restrict Access to Particular File.

the file to which attempt is made to access (open) at run-time. Event `initialize` is specified so that the set (and other variables, if present) can be initialized. An excerpt of the PEDL script for example Restrict Access to Particular File is given in Figure 5.5 .

On the MEDL side, event `fileOpened` is imported into the script for requirement specification. With the value of the String method communicated to the Run-time Checker, operation `value(e,0)` gives the value of the first input parameter of event `e`, provided `e` occurs. Whenever a file access is attempted on file `passwd`, an alarm will sound.

An excerpt of the MEDL script for example Restrict Access to Particular File is given in Figure 5.6. Alarm `exceedingAllowableValue` is raised whenever there is attempt to open file `passwd`.

```
var String valueFileOpened;
var int filePASSOpenedVar;

event filePASSOpened = start(valueFileOpened==‘‘passwd’’);

alarm filePASSOpenedVarAlarm = start(filePASSOpenedVar >= 1);

filePASSOpened -> {
    filePASSOpenedVar’ = filePASSOpenedVar + 1;
}

initialize -> {
    filePASSOpenedVar’ = 0;
    numFileOpened’ = 0;
}

fileOpened -> {
    valueFileOpened’ = value(fileOpened,0);
}
```

Figure 5.7: Excerpt of MEDL Script for Example Restrict File Access Amount.

5.1.5 Example: Restict File Access Amount

This last example serves to exemplify the expansive reach of resource-aware policies that are now possible with MaC. It illustrates a policy that specifies that certain files can only be accessed a certain number of times. More specifically, this policy specifies that the file named `passwd` can only be opened once. Each time a file is opened, event `fileOpened`, which indicates the opening of a file, registers the filename into variable `valueFileOpened`. Event `filePASSOpened` is recognized as occurring, as soon as variable `valueFileOpened` equals the filename `passwd`. Each time the event `filePASSOpened` occurs, a counter `filePASSOpenedVar` variable is incremented by one. When the number of times variable `filePASSOpenedVar` (or similarly, the number of times filename `passwd` is opened) equals or exceeds one, alarm `filePASSOpenedVarAlarm` sounds.

An excerpt of the MEDL script for example Restrict File Access Amount is given in Figure 5.7. If an attempt is made to open a file named `passwd`, event `filePASSOpened` occurs, which increments variable `filePASSOpenedVar` by one. When this variable equals or exceeds one (i.e. attempt is made to open file `passwd` at any time), alarm `filePASSOpenedVarAlarm` will sound.

The above five examples illustrate a selection of the resource-aware policies that MaC is now able to enforce, given the extensions described in Chapter 4. These examples illustrate that it is possible to enhance the security of Java programs using run-time monitoring. Although Java monitors *access* to system resource use, it does not monitor the *extent* of use. The above five examples serve to exemplify some of the ways the MaC software can now be used to specify and enforce desired resource limits on downloaded applets.

The examples provided in this chapter are not exhaustive and serve only to illustrate how the extensions presented in previous Chapter 4 can now be exercised to develop resource-aware policies. Note that this approach could also be used to monitor and control usage of other resources and services provided by the Java API. For example, other resources include:

- Network communication
- User input
- Database drivers
- Distributed computation through Java Remote Method Interface (RMI) or

Common Object Request Broker Architecture (CORBA)

- ... and many others

Chapter 6

Summary and Future Work

6.1 Summary

We have presented an architecture for enforcing resource-aware policies using run-time formal analysis. An existing approach to the monitoring and checking of running systems, Monitoring and Checking (MaC) software, was extended to allow the specification and enforcement of desirable constraints on resource consumption. Specifically, our work focused on resource awareness of file system use.

The main contribution of this thesis is confirming that the enforcement of resource-aware policies can be achieved with run-time formal analysis. This is demonstrated by extending and implementing a prototype system of MaC with our extensions. More specifically, the Monitoring and Checking (MaC) architecture was extended with the following abilities:

1. Monitor Java API methods

2. Allow method arguments to be communicated

3. Allow Strings to be communicated
4. Add sets to MEDL
5. Perform set operations `emptySet`, `add`, `size`, and `print`

A central aspect of the MaC architecture is its use of formal requirement specifications to check the run-time execution of a target program. In our research project, we implemented a MaC prototype with our extensions, and showed the additional security of run-time formal analysis for enforcing resource-aware policies, through several examples.

In addition, extensions provided in this research project have a general availability; that is, they facilitate the use of MaC in general, and not just in the context of monitoring resource usage.

6.2 Future Work

6.2.1 Short Term

The extensions provided in this thesis can be exercised to develop and enforce certain kinds of resource-aware policies with the MaC software. A good selection of policies are possible. However, there are some natural extensions to this work that would help expand the range of policies MaC can enforce; these extensions can be drawn out from the limitations provided in Chapter 4.

6.2.2 Future Work

Probing deeper, this research project also provides a strong foundation for future work that focusses on the enforcement of resource-aware policies using run-time analysis. This subsection discusses the following four directions:

1. Alternative modes of communication

2. Logical frameworks expansion

3. Feedback capability extension

4. Application to other languages

6.2.3 Alternative Modes of Communication

An additional area of research is the exploration into appropriate modes of communication. Unrestricted object communication is important for monitoring system resource use. Without the ability to communicate more complex objects other than Strings, the kinds of policies MaC can enforce is limited. For example, without the ability to communicate file descriptors, it is not possible to specify a policy that limits the number of files *currently* open. A file descriptor is a handle that uniquely identifies a file – the action of closing a file is performed on its file descriptor [Mic03b]. As a result, the same file could be closed many times, but without the ability to recognize its file descriptor, MaC could only assume that different files are being closed and cannot identify the closing of any one file in particular.

The current mode of communication is adequate for a number of resource-aware policies, but may not be the best choice. This is especially clear from the current technique in which communication is done through Data Streaming over Pipe Streaming.

Data Streaming communication includes primitive Java data types as well as String objects [Mic03b]. A Data Streaming approach to communication recognizes basic data types: *boolean, byte, char, short, int, long, float, double*, and also *String* types which are necessary for filename communication. However, Object Streaming communication supports primitive data types as well as all objects, including Strings, file descriptors, etc. As a result, more interesting resource-aware policies are possible with communication done through Object Streaming. Certain actions on system resources, such as file close, are performed on a class object (in this case, a file descriptor), that uniquely identifies that resource. A first step in looking at alternative modes of communication would require a more stable release of Java and research into how to implement communication via Object Streaming over Pipes in MaC.

6.2.4 Logical Frameworks Expansion

Different requirements can be best expressed using different underlying logics, so an avenue for future research is to explore the attachment of new logical frameworks for expressing policies with MaC. Adding other logics to MaC would increase its coverage of monitoring and enhance the expressiveness of monitoring scripts. Currently, MaC uses a kind of interval past-time temporal logic of events to specify program behaviours. However, policy range could be expanded if MaC were to try to use other kinds of logics in its policy specification. For example, future-time logics could enable the specification of properties that prevent, rather than detect violation once it has occurred. Similarly, UML notations could be used to check original designs (via state charts and/or sequence diagrams) against execution traces.

6.2.5 Feedback Capability Extension

This work has extended MaC to the enforcement of resource-aware policies. Although MaC can now be used to detect violations of aspects of desired file resource consumption, MaC is unable to provide any feedback to the running system. A remedy for this weakness exists for the original MaC implementation. The resulting system is called MaCS (Monitoring and Checking with Steering). The feedback component uses the information collected during monitoring to steer the application back to a safe state after a violation occurs [KLS+02].

It would be desirable to extend this research project with a feedback capability. In our case, MaCS could detect violations of desired limits of applet resource consumption, and implement a remedy for the situation. More specifically, the current MaCS approach involves specification of an additional "steering script" which defines steering actions. An avenue of future research would involve applying this research project to the MaCS approach.

6.2.6 Application to other Languages

The MaC system has been applied to the monitoring of Java bytecode programs; MaC instrumentation is performed on Java bytecodes. As a result, our extensions to MaC are in Java. However, it is worthwhile investigating the methodology of applying the MaC architecture to support various target platforms other than Java. The popularity of the Web has lead to more regular use of Java *and* many other languages that support mobility. Although Java now easily tops the list of preferred platforms for Internet-savvy mobile code, other mobile code languages are gaining in popularity.

Bibliography

[Ais03] Selim Aissi. Runtime environment security models. *Intel Technology Journal*, 07(01), 2003.

[atUoP] Real-Time Systems Group at the University of Pennsylvania. Introduction to Trek class library. Web. http://www.cis.upenn.edu/˜rtg/mac/jtrek-doc/overpack.html.

[CMS01] Ajay Chander, John C. Mitchell, and Insik Shin. Mobile code security by Java bytecode instrumentation. In *Proceedings of the 2001 DARPA Information Survivability Conference and Exposition (DISCEX II)*, June 2001.

[CvE98] Grzegorz Czajkowski and Thorsten von Eicken. Jres: A resource accounting interface for Java. In *Proceedings of the 1998 ACM OOPSLA Conference*, October 1998.

[CW96] Mary Campione and Kathy Walrath. *The Java Tutorial.* Addison-Wesley, 1996.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification.* Sun Microsystems, Inc., second edition, 2000.

[Her02] Almut Herzog. *Secure Execution Environment for Java Electronic Services.* PhD thesis, Department of Computer and Information Science, Linkoping University, 2002.

[HK02] J. Hulaas and D. Kalas. Monitoring of resource consumption in Java-based application servers. In *Proceedings of the 10th HP Openview University Association Plenary Workshop (HP-OVUA'2003)*, 2002.

[Hs01] Klaus Havelund and Grigore Roşu. Monitoring Java programs with Java PathExplorer. In *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier, 2001.

[Hs02a] Klaus Havelund and Grigore Roşu. Synthesizing monitors for safety properties. In *Proceedings of International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2002.

[HS02b] Almut Herzog and Nahid Shahmehri. Using the Java sandbox for resource control. In *Proceedings of the 7th Nordic Workshop on Secure IT Systems (NordSec)*, pages 135–147, November 2002.

[Kim99] Moonjoo Kim. *Information Extraction for Run-time Formal Analysis.* PhD thesis, Department of Computer Science, University of Pennsylvania, 1999.

[KKL$^+$02] Moonjoo Kim, Sampath Kannan, Inpup Lee, Oleg Sokolsky, and Mahesh Viswanathan. Computational analysis of run-time monitoring – fundamentals of Java-MaC. In *2nd International Workshop on Run-time Verification*, Copenhagen, Denmark, July 26, 2002.

[KKLS01] Moonjoo Kim, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Java-MaC: a run-time assurance tool for Java. In *1st International Workshop on Run-time Verification*, Paris, France, July 23, 2001.

[KLS$^+$02] Moonjoo Kim, Insup Lee, Usa Sammapun, Jangwoo Shin, and Oleg Sokolsky. Monitoring, checking, and steering of real-time systems. In *2nd International Workshop on Run-time Verification*, Copenhagen, Denmark, July 26, 2002.

[KVBA$^+$98] Moonjoo Kim, Mahesh Viswanathan, Hanene Ben-Abdallah, Sampath Kannan, Insup Lee, and Oleg Sokolsky. MaC: A framework for run-time correctness assurance of real-time systems. Technical report, MS-CIS-98-37, Department of Computer and Information Sciences, University of Pennsylvania, December 1998.

[LBAK$^+$98] Insup Lee, Hanene Ben-Abdallah, Sampath Kannan, Moonjoo Kim, Oleg Sokolsky, and Mahesh Viswanathan. A monitoring and checking framework for run-time correctness assurance. In *Proceedings of the 1998 Korea-U.S. Technical Conference on Strategic Technologies*, Vienna, VA, October 22-24, 1998.

[LBAK$^+$99] Insup Lee, Hanene Ben-Abdallah, Sampath Kannan, Moonjoo Kim, Oleg Sokolsky, and Mahesh Viswanathan. Formally specified monitoring of temporal properties. In *European Conference on Real-Time Systems*, York, UK, June 9-11, 1999.

[Lin03] Clark S. Lindsey. Java tech. Web, 2003. http://www.particle.kth.se/~lindsey/JavaCourse/Book/index.html.

[LKK$^+$99] Insup Lee, Sampath Kannan, Moonjoo Kim, Oleg Sokolsky, and Mahesh Viswanathan. Runtime assurance based on formal specifications. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, Monte Carlo Resort, Las Vegas, Nevada, USA, June 28-July 1, 1999.

[Mar03] Kevin Marron. Bolster defences, computer users urged. *The Globe and Mail, Globe E-business*, page E1, March 2003.

[MF99] Gary McGraw and Edward Felten. *Securing JAVA*. John Wiley and Sons, Inc., 1999.

[Mic02] Sun Microsystems. JavaCC: JJTree reference documentation. Web, 2002. http://javacc.dev.java.net/doc/JJTree.html.

[Mic03a] Sun Microsystems. The Java tutorial. Web, 2003. http://java.sun.com/docs/books/tutorial/index.html.

[Mic03b] Sun Microsystems. JavaTM 2 platform, standard edition, v 1.4.2 api specification. Web, 2003. http://java.sun.com/j2se/1.4.2/docs/api/index.html.

[MS98] N. V. Mehta and K. R Sollins. Expanding and extending the security features of Java. In *Proceedings of the 7th USENIX Security Symposium*, pages 159–172, January 1998.

[RW02] Algis Rudys and Dan S. Wallach. Enforcing Java run-time properties using bytecode rewriting. In *Proceedings of the International Symposium on Software Security*, November 2002.

[SBB$^+$00] Niranjan Suri, Jeffrey M. Bradshaw, Maggie R. Breedy, Paul T. Groth, Gregory A. Hill, Renia Jeffers, and Timothy S. Mitrovich. An overview of the nomads mobile agent system. In Ciar and Bryce, editors, *6th ECOOP Workshop on Mobile Object Systems*, volume 13, June 2000.

[VB01] Alex Villazón and Walter Binder. Portable resource reification in Java-based mobile agent systems. In *Proceedings of The Fifth IEEE International Conference on Mobile Agents (MA'01)*, December 2001.

[Ven97] Bill Venners. Java security: How to install the security manager and customize your security policy. *JavaWorld, Under The Hood*, page 1, November 1997.

[Wel03] Jason Wells. Death to final! Web, 2003. http://crispyneurons.com/writing/death_to_final.pdf.

[Yak02] Vladimir Omar Calderon Yaksic. J-RAF – the Java resource accounting facility. Master's thesis, Département d'informatique, Faculté des Sciences, Université De Genève, 2002.

Appendix A

BNF for PEDL

addObj	::= *java code*
location	::= *java code*
checkMethodObj	::= *java code*
contained	::= *java code*
addExec	::= *java code*
setRelOp	::= *java code*
setArithOp	::= *java code*
addToSymbolTable	::= *java code*
searchSymbolTable	::= *java code*
connectAndSetName	::= *java code*
error_skipto	::= *java code*
MonitoringScript	::= <MONSCR> <IDENTIFIER> Declarations Definitions <END>
Declarations	::= (((EventDeclaration | ConditionDeclaration | MonitoredObjectDeclaration | MonitoredMethodDeclaration)) (";")?)*

```
                  Definitions  ::= ( ( ( EventDefinition |
                               ConditionDefinition ) ) ( ";" )?  )*
             EventDeclaration  ::= <EXPORT> <EVENT> EventNames
         ConditionDeclaration  ::= <EXPORT> <CONDITION> ConditionNames
                   EventNames  ::= <IDENTIFIER> ( "," <IDENTIFIER> )*
               ConditionNames  ::= <IDENTIFIER> ( "," <IDENTIFIER> )*
 MonitoredObjectDeclaration  ::= <MONOBJ> Type ( AliasVariableName
                               "<-" )?  ObjectName ( "," ( <IDENTIFIER>
                               "<-" )?  ObjectName )*
 MonitoredMethodDeclaration  ::= <MONMETH> ( Type | <VOID> ) (
                               AliasMethodName "<-" )?  MethodName
                   ObjectName  ::= ( <IDENTIFIER> "." )* <IDENTIFIER>
                               ( "." <IDENTIFIER> FormalParameters )?
                               "." <IDENTIFIER>
     MethodNameWOparameters  ::= ( <IDENTIFIER> "." )+ <IDENTIFIER>
                   MethodName  ::= MethodNameWOparameters
                               FormalParameters
            AliasVariableName  ::= <IDENTIFIER>
              AliasMethodName  ::= <IDENTIFIER>
             FormalParameters  ::= "(" ( Type ( "," Type )* )?  ")"
              EventDefinition  ::= <EVENT> <IDENTIFIER> "="
                               EventExpression
              EventExpression  ::= ( SimpleEventExpression ( ( EventOp
                               EventExpression ) )?  OptionalWhen )
                 OptionalWhen  ::= ( <WHEN> ConditionalExpression )*
        SimpleEventExpression  ::= ( <IDENTIFIER> | "(" EventExpression
                               ")" | <START> "(" ConditionalExpression
                               ")" | <END> "(" ConditionalExpression
                               ")" | <UPDATE> "(" ObjectName ")" |
                               <STARTM> "(" MethodNameWOparameters
                               FormalParameters ")" | <ENDM> "("
                               MethodNameWOparameters FormalParameters
                               ")" | <IOM> "(" MethodNameWOparameters
                               FormalParameters ")" )
               EventOp  ::= "&&" | "||"
   ConditionDefinition  ::= <CONDITION> <IDENTIFIER> "="
                            ConditionalExpression
                  Type  ::= ( ( PrimitiveType | Name ) ( "[" "]"
                            )* )
         PrimitiveType  ::= ( "boolean" ) | ( "char" ) | (
                            "byte" ) | ( "short" ) | ( "int" ) |
                            ( "long" ) | ( "float" ) | ( "double" )
                            | ( "String" )
                  Name  ::= <IDENTIFIER> ( "." ( <IDENTIFIER> )
                            )*
              NameList  ::= Name ( "," Name )*
            Expression  ::= ConditionalExpression
 ConditionalExpression  ::= ConditionalOrExpression
ConditionalOrExpression ::= ( ConditionalAndExpression ( "||"
                            ConditionalAndExpression )* )
ConditionalAndExpression ::= ( InclusiveOrExpression ( "&&"
                            InclusiveOrExpression )* )
 InclusiveOrExpression  ::= ( ExclusiveOrExpression ( "|"
                            ExclusiveOrExpression )* )
 ExclusiveOrExpression  ::= ( AndExpression ( "^"  AndExpression
                            )* )
         AndExpression  ::= ( EqualityExpression ( "&"
                            EqualityExpression )* )
    EqualityExpression  ::= ( <INM> "(" MethodNameWOparameters
                            FormalParameters ")" ) | ( <DEFINED>
                            "(" ConditionalExpression ")" ) |
                            InstanceOfExpression ( ( ( "==" | "!=" )
                            InstanceOfExpression ) )*
  InstanceOfExpression  ::= ( RelationalExpression (
                            "instanceof" Type )?
  RelationalExpression  ::= ShiftExpression ( ( ( "<" | ">" |
                            "<=" | ">=" ) ShiftExpression ) )*
       ShiftExpression  ::= AdditiveExpression ( ( ( "<<" | ">>"
                            | ">>>" ) AdditiveExpression ) )*
```

```
        AdditiveExpression  ::= MultiplicativeExpression ( ( ( "+" |
                                "-" ) MultiplicativeExpression ) )*
  MultiplicativeExpression  ::= UnaryExpression ( ( ("*" | "/" | "%"
                                ) UnaryExpression ) )*
           UnaryExpression  ::= ( ( "+" | "-" ) UnaryExpression ) |
                                UnaryExpressionNotPlusMinus | ( <TIME>
                                "(" EventExpression ")" ) | ( <VALUE>
                                "(" EventExpression "," Expression ")" )
UnaryExpressionNotPlusMinus ::= ( ( " " | "!" ) UnaryExpression ) |
                                CastExpression | PrimaryExpression
             CastLookahead  ::= "(" PrimitiveType | "(" Name "["
                                "]" | "(" Name ")" ( " " | "!" | "(" |
                                <IDENTIFIER> | Literal )
            CastExpression  ::= ( "(" Type ")" UnaryExpression | "("
                                Type ")" UnaryExpressionNotPlusMinus )
         PrimaryExpression  ::= PrimaryPrefix ( ( PrimarySuffix ) )*
             PrimaryPrefix  ::= Literal | "("Expression")" |
                                ObjectName | <IDENTIFIER>
             PrimarySuffix  ::= ( "[" Expression "]" | "."
                                <IDENTIFIER> | Arguments )
                   Literal  ::= ( <INTEGER_LITERAL> )| (
                                <FLOATING_POINT_LITERAL> ) |
                                ( <CHARACTER_LITERAL> ) | (
                                <STRING_LITERAL> ) | ( "true" ) | (
                                "false" ) | ( "null" )
                 Arguments  ::= "(" ( ArgumentList )? ")"
              ArgumentList  ::= Expression (","Expression)*
```

Appendix B

BNF for MEDL

setRelOp	::= *java code*
setArithOp	::= *java code*
addToSymbolTable	::= *java code*
searchSymbolTable	::= *java code*
error_skipto	::= *java code*
RequirementScript	::= <REQSCR> <IDENTIFIER> Statements Guards <END>
Statements	::= ((Statement) (";")?)*
Statement	::= EventDeclaration | ConditionDeclaration | ActionDeclaration | AuxiliaryVariableDeclaration | EventDefinition | ConditionDefinition | SafetyPropertyDefinition | AlarmDefinition
EventDeclaration	::= <IMPORT> <EVENT> IdentifierList
ConditionDeclaration	::= <IMPORT> <CONDITION> IdentifierList
ActionDeclaration	::= <IMPORT> <ACTION> IdentifierList
IdentifierList	::= <IDENTIFIER> ("," <IDENTIFIER>)*
AuxiliaryVariableDeclaration	::= <AUXVAR> PrimitiveType IdentifierList
EventDefinition	::= <EVENT> <IDENTIFIER> "=" EventExpression

```
     ConditionDefinition  ::= <CONDITION> <IDENTIFIER> "="
                              ConditionalExpression
SafetyPropertyDefinition  ::= <PROPERTY> <IDENTIFIER> "="
                              ConditionalExpression
         AlarmDefinition  ::= <ALARM> <IDENTIFIER> "="
                              EventExpression
         EventExpression  ::= ( SimpleEventExpression ( ( EventOp
                              EventExpression ) )?  OptionalWhen )
            OptionalWhen  ::= ( <WHEN> ConditionalExpression )*
   SimpleEventExpression  ::= ( <IDENTIFIER> | "(" EventExpression
                              ")" | <START> "(" ConditionalExpression
                              ")" | <END> "(" ConditionalExpression
                              ")" )
                 EventOp  ::= "&&" | "||"
                  Guards  ::= ( Guard )*
                   Guard  ::= <IDENTIFIER> "->" "{" Updates "}"
                 Updates  ::= Update (";" Update)*";"
                  Update  ::= ( (<IDENTIFIER>> "=" Expression |
                              ( <INVOKE> <IDENTIFIER>) | (<PRINT>
                              Expression) )
                    Type  ::= ( (PrimitiveType|Name)("[" "]")* )
           PrimitiveType  ::= ( "boolean" ) | ( "char" ) | (
                              "byte" ) | ( "short" ) | ( "int" ) |
                              ( "long" ) | ( "float" ) | ( "double" )
                              | ( "String" ) | ( "Set" )
              ResultType  ::= ( "void" ) | ( Type )
                    Name  ::= <IDENTIFIER> ( "." ( <IDENTIFIER> )
                              )*
              Expression  ::= ConditionalExpression
                              |SetExpression
           SetExpression  ::= SimpleSetExpression
       sizeSetExpression  ::= sizeSimpleSetExpression
     SimpleSetExpression  ::= ( <SETADD> ”(” Expression ”,”
                              SetExpression | <SETEMPTY> |
                              <IDENTIFIER> )
```

sizeSimpleSetExpression	::= ***Name***
ConditionalExpression	::= ConditionalOrExpression ("?" Expression ":" ConditionalExpression)?)
ConditionalOrExpression	::= (ConditionalAndExpression ("||" ConditionalAndExpression)*)
ConditionalAndExpression	::= (InclusiveOrExpression ("&&" InclusiveOrExpression)*)
InclusiveOrExpression	::= (ExclusiveOrExpression ("|" ExclusiveOrExpression)*)
ExclusiveOrExpression	::= (AndExpression ("^" AndExpression)*)
AndExpression	::= (EqualityExpression ("&" EqualityExpression)*)
EqualityExpression	::= ("[" EventExpression ";" EventExpression ")") | (<DEFINED> "(" Expression ")") | InstanceOfExpression ((("==" | "!=") InstanceOfExpression))*
InstanceOfExpression	::= (RelationalExpression ("instanceof" Type)?
RelationalExpression	::= ShiftExpression ((("<" | ">" | "<=" | ">=") ShiftExpression))*
ShiftExpression	::= AdditiveExpression ((("<<" | ">>" | ">>>") AdditiveExpression))*
AdditiveExpression	::= MultiplicativeExpression ((("+" | "-") MultiplicativeExpression))*
MultiplicativeExpression	::= UnaryExpression ((("*" | "/" | "%") UnaryExpression))*
UnaryExpression	::= (("+" | "-") UnaryExpression) | (***<SETSIZE> "(" sizeSetExpression ")")*** | UnaryExpressionNotPlusMinus | (<TIME> "(" EventExpression ")") | (<VALUE> "(" EventExpression "," Expression ")") | (***<SETADD> "(" Expression "," SetExpression ")")***

```
UnaryExpressionNotPlusMinus  ::= ( ( " " | "!" ) UnaryExpression ) |
                                 CastExpression | PrimaryExpression
             CastLookahead  ::= "(" PrimitiveType | "(" Name "["
                                 "]" | "(" Name ")" ( " " | "!" | "(" |
                                 <IDENTIFIER> | Literal )
            CastExpression  ::= ( "(" Type ")" UnaryExpression | "("
                                 Type ")" UnaryExpressionNotPlusMinus )
         PrimaryExpression  ::= PrimaryPrefix ( ( PrimarySuffix ) )*
             PrimaryPrefix  ::= SetLiteral | Literal |
                                 "("Expression")" | ResultType "."
                                 "class" | Name
             PrimarySuffix  ::= ( "[" Expression "]" | "."
                                 <IDENTIFIER> | Arguments )
                   Literal  ::= ( <INTEGER_LITERAL> )| (
                                 <FLOATING_POINT_LITERAL> ) |
                                 ( <CHARACTER_LITERAL> ) | (
                                 <STRING_LITERAL> ) | ( "true" ) | (
                                 "false" ) | ( "null" )
                SetLiteral  ::= <SETEMPTY>
                 Arguments  ::= "(" ( ArgumentList )? ")"
              ArgumentList  ::= Expression (","Expression)*
```

Vita

Name	Natalie Alexandra Bowles
Place and year of birth	Ottawa, Ontario, Canada, 1978
Education	Queen's University, Kingston, Ontario, 2002-2004 M.Sc. (Computing and Information Science), 2004
	Queen's University, Kingston, Ontario, 1998-2002 B.Sc. (Honours, Computing and Information Science), 2002
	Queen's University, Kingston, Ontario, 1997-2001 B.Com (Honours), 2001
Experience	Research assistant, School of Computing, Queen's University, 2002-2004
Awards	Office of Critical Infrastructure Protection and Emergency Preparedness (OCIPEP) Research Fellowship, 2002-2003 Queen's Graduate Award (QGA) , 2002-2003 Queen's University, School of Business, Dean's Honour Role, 1997-1998 D. I. McLeod Scholarship, 1998 Lena MacNeil Scholarship, 1997 NORTEL Scholarship, 1997 Student Council Scholarship, 1997