Poznań University of Technology
Faculty of Computer Science and Management
Institute of Computer Science

Master's thesis

DESIGN AND IMPLEMENTATION OF A JAVA SOURCE CODE PRECOMPILATION TOOL FOR STATIC ANALYSIS AND MODIFICATION OF PROGRAMS FOR THE ATOMIC RMI LIBRARY

Konrad Siek, 71747

Supervisor
dr hab. inż. Paweł T. Wojciechowski

Poznań, 2009

Contents

Chapter 1

Introduction

1.1 Motivations

In consequence of the unpredictable manner in which concurrent data access of shared memory in distributed systems is performed, a correct result of such processing cannot be guaranteed. Thus, concurrency control is an important aspect of almost all distributed processing; and, therefore, it is usual for modern programming languages to supply a number of constructs allowing or facilitating the synchronization of concurrent processes[Sun08b], including critical sections, monitors, locks, conditional variables and semaphores.

Nonetheless, these low-level mechanisms do not provide the developer with aid in terms of identifying sections of code that require synchronization, nor do they assist in managing the blocking mechanisms – both being especially error-prone areas. In addition, programs written using the aforementioned simple mechanisms cannot be directly re-used, due to their lack of compositionality[HMJH06], which is the assurance that the safety of separate components guarantees the safety of their combination.

The Atomic RMI[Mam08, Woj08, Woj07, WRS04] library proposes to solve those problems by turning to the idea of Software Transactional Memory: Atomic RMI allows the declaration of *tasks* (similar to transactions known from Database Management Systems) whose correct processing is enforced by the system, thus ensuring synchronization. The result of the use of tasks and STM in Atomic RMI, is lifting the chore of manual management of blockades from the developer, therefore reducing his efforts to declaring transactions.

However, the execution of various tasks is performed by Atomic RMI with a diminished degree of concurrency, unless the value of maximum number of method calls issued to each remote object in that task is provided beforehand. This requirement creates the necessity for the developer to analyze the code for each task and manually establish the upper bound on method calls to remote objects. Following that, the need arises for him to write additional code which invokes the appropriate methods of the Atomic RMI API that inform the system of these numbers. Therefore, an additional burden is placed on the developer.

In the face of the developer's additional task, this work explores the possibility of creating a tool for the automation of the process of analyzing source code to establish upper bounds on the number of remote object calls and generating task headers.

1.2 Goals and requirements

The main goal of this thesis is to create a tool that would be able to analyze source code of a program designed to run with the Atomic RMI library and discern, as far as possible, the number of times each remote object is called by any of its constituent methods. Afterward, the tool should be capable of modifying the source code of that program for each of the Atomic tasks to include information about the maximum number of possible calls to remote objects in the way defined by the Atomic RMI library.

The achievement of the primary objective requires the following actions to be conducted, each serving to fulfill a particular sub-goal:

1. Experimentation with parser/lexer generators for the Java Language: JFlex/CUP, JavaCC, SableCC, ANTLR, and consideration of their capabilities, shortcomings, and available resources, in order to discern which of their features are essential to the implementation of the precompilation tool, and select the most apt generator.

2. Familiarization with methods for static program analysis, and code verification algorithms, with the intention of use them in the implementation stage of the project.

3. Implement a Java code precompilation tool for the analysis and modification of programs using the Atomic RMI library, fulfilling two main requirements:

    - Analyze the source code of a program and evaluate an upper bound on the number of calls to remote objects.
    - Provide headers to Atomic tasks by generation of appropriate additional source code to the program.

    Specific requirements and their rationale are presented in 2.3.

1.3 Structure

This paper is structured as follows. After the current chapter explains the motivations, goals, and the structure of this paper, chapter 2 overviews the systems that inter-operate with the tool implemented as a result of this work, and describes the requirements for their co-operation. Next, chapter 3 details the algorithm used to determine the upper bound on the number of method calls to objects, as well as Simple Language — a simplified version of the Java language for which the algorithm is designed. Following that, chapter 4 provides a summary description of popular parser/lexer generators; additionally, a short glossary of the terms used in static analysis is given. Chapter 5 presents the process of development stage by stage, as well as the architecture and possible configuration of the precompilation tool. Chapter 6 enumerates the conducted tests and describes their outcomes. Finally, chapter 8 summarizes the results of the thesis and points out directions for future development. In addition, appendix A provides a complete EBNF definition of the Simple Language.

Chapter 2

Context

This chapter describes the context of the Java precompilation tool, and the systems for cooperation with which it is primarily designed.

The first two sections describe Java RMI and Atomic RMI, reiterating their advantages and briefly describing their methods of operation. The final section presents the tool developed alongside this thesis and the way it is integrated with Atomic RMI.

2.1 Java RMI

Java RMI[Sun08a] (Java Remote Method Invocation) is an extension to the Java language that allows programs to access methods of objects that are defined in other Java Virtual Machines (JVM). The local JVM uses stub objects to interface with objects, which may be implemented and executed on a remote JVM.

Java RMI resembles Remote Procedure Call mechanisms, however it provides a number of noteworthy advantages over the traditional approach, of which it is essential to highlight the following,

- implicit serialization of objects that are passed to remote methods as arguments,
- passing behavior definition between remote and local objects,
- a built-in security mechanism for code retrieval,
- reference passing between JVMs, and a distributed garbage collection mechanism,
- automatic class stub generation.

The aforementioned characteristics allow for a seamless integration of remote objects into local code, rendering the creation of distributed systems more intuitive for the developer. However, the creation of distributed systems necessitates that various additional aspects be considered, including synchronization of concurrent processes, which are ubiquitous in such systems.

It must be noted that Java RMI does not aim to supply any additional means of synchronization, apart from those available in the Java language by default. Specifically, Java RMI does not ensure that the execution of concurrent processes is serializable. Therefore, it is not guaranteed that the effect of the execution (i.e. the state of the system) of two or more concurrent processes will be equivalent to the effect of a sequential execution of those two processes. In consequence, it is the developer who needs to ensure serializability by implementing a synchronization policy.

2.2 Atomic RMI

Atomic RMI[Mam08, Woj08, Woj07, WRS04] is a rollback-free concurrency control library built on top of Java RMI, which ensures serializable execution of concurrent processes by allowing the developer to define specific sequences of remote method calls as atomic sequences.

Such definition of atomic sequences, called atomic tasks, is not unakin to multi-threaded transactions decomposed to ensure the property of isolation (also known as linearizability[HW90]), which requires the concurrent execution of such tasks to be equivalent to an execution in which the tasks would be executed sequentially. However, in contrast to traditional language facilities (such as locks, semaphores, and monitors) atomic tasks are able to perform developer-defined operations. In deference to the fact that arbitrary operations may include irrevocable effects (such as those cause by input and output) the order of critical operation of atomic tasks is tightly controlled, therefore providing a guarantee that once started they cannot run into conflicts requiring them to be rolled back.

The main premise for the operation of Atomic RMI is a runtime locking strategy for the aforementioned atomic tasks. In essence, task threads are assigned version counters – tickets – which enable them to acquire access to resources. At the entry point of each task, its thread obtains incremented ticket values – versions – for all the resources that it plans to access in the course of its operation, and the thread is able to access a resource only in the case when the thread's ticket count and the corresponding resource's internal counter are equal; in other cases, the thread is delayed. In consequence, since a version is generated automatically, and thus unique for all tasks, it may serve to obtain exclusive access to a resource; furthermore, the ordering of the versions causes atomic tasks to access the shared resources in an order preserving atomicity.

To be precise, the Atomic RMI library enforces serializability through an application of one of the three versioning algorithms (introduced in [WRS04]):

- BVA – Basic Versioning Algorithm,
- SVA – Supremum Versioning Algorithm,
- CVA – Composite Versioning Algorithm.

Among those algorithms, all of them require knowledge about the course of the system's operation to be provided in advance of its execution: SVA requires the preambles (headers) of all tasks to specify beforehand the maximum number of calls that may be issued to each remote object as a result of execution of that task. And even though CVA is able to operate despite that requirement, it achieves an improved degree of concurrency if the upper bound on the number of calls to specific remote objects is known in advance, and otherwise falls back on the less efficient mode of operation used by BVA.

Therefore, to obtain maximum efficiency from Atomic RMI, the developer must provide the maximum number of calls to the most complete set of remote objects possible, by first examining the source code of the task and then providing the numbers using the Atomic RMI library API. To maintain correctness over time, this should be repeated with each change to the source code.

Among the most important advantages of building Atomic RMI on top Java RMI is that no dedicated implementation of Java RMI needed to be created, and the development of Atomic RMI could be successfully concentrated on the realization of complex function (i.e. concurrency control).

The Atomic RMI library extended the standard implementation of Java RMI by adding a number of additional elements, which intercept all calls to remote objects (both client- and server-side) in order to check whether the call may be executed instantly, or whether it is necessary to delay it until the version number of the atomic task and the remote service number match.

The fact that the first phase of the atomic task's evaluation requires the incrementation of global remote resources' service counters to be executed as a part of the critical section, along with other atomic tasks, a central governing service called the Atomic Task Manager was introduced to the Atomic RMI library. This entity is responsible for managing the service counters pertaining to remote resources, coordinating tasks' initial and final phases with reference to incrementation of the aforementioned counters, and failure detection of client processes.

In effect, the architecture of an extremely simple application using the Atomic RMI library, such as that presented in figure 2.1 must contain an Atomic Task Manager, and at least one of each: a server process, a client process, and an RMI registry. It is also important to note, that the processes may be placed in any combination of separate or shared Java Virtual Machines.

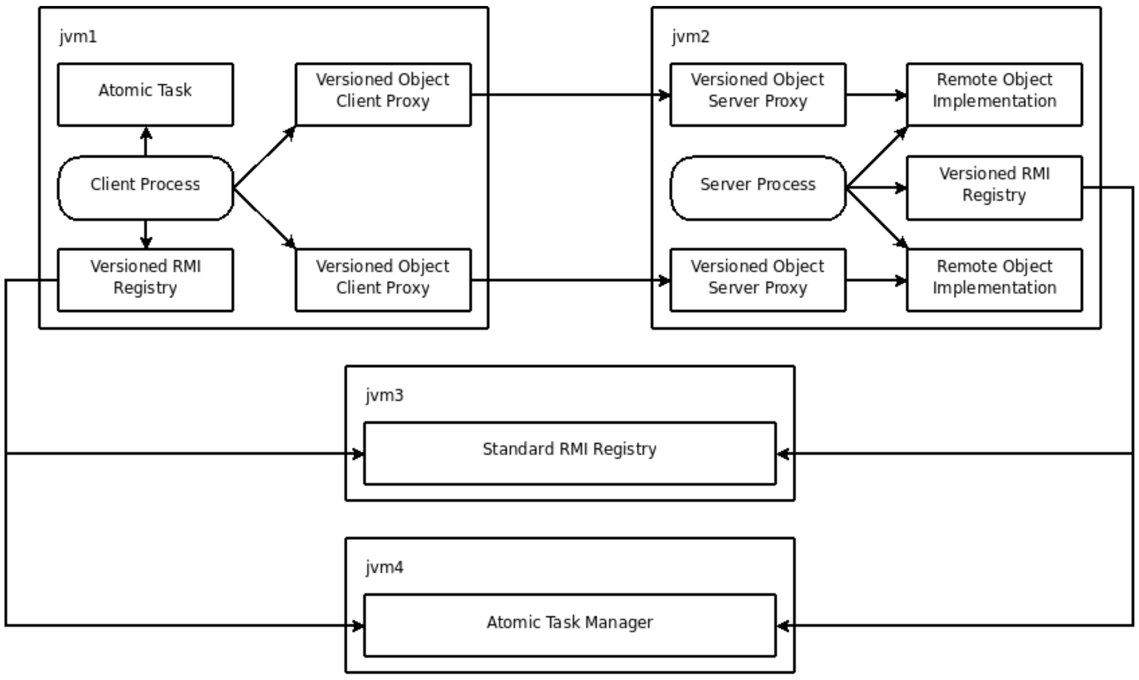
jvm1
Atomic Task
Versioned Object Client Proxy
Client Process
Versioned RMI Registry
Versioned Object Client Proxy
jvm2
Versioned Object Server Proxy
Remote Object Implementation
Server Process
Versioned RMI Registry
Versioned Object Server Proxy
Remote Object Implementation
jvm3
Standard RMI Registry
jvm4
Atomic Task Manager

FIGURE 2.1: An example of an architecture of a system built using Atomic RMI.

An example of a simple client program is shown in listing 2.2. It implements two concurrent tasks (defined as blocks to avoid generation of irrelevant code): the first task performs a transfer between account – an `amount` is withdrawn from `accountA` twice, and the total is deposited to `accountB`; the second task checks the balance on both accounts and prints their joint total, which should be equal to the total amount of assets initially deposited on both accounts. The act of displaying a message has an irrevocable effect – once performed, it cannot be undone.

The example presents a situation where the property of isolation must be ensured in order to maintain a consistent view of the bank assets; otherwise, it would be possible for the second task to obtain all its information after the transfer has commenced, but before its completion (the lines 46, 47 of the second task would be executed after line 25, but before line 27), causing the asset total to be incorrect by the value of up to twice `amount`. Moreover, since the implementation of atomic tasks does not involve rollbacks, the balance is never printed out more than once.

From the developer's perspective, the creation of a client-side Atomic RMI requires the following operations to be performed:

1. Access must be gained to a centralized Atomic Task Manager (line 2),
2. An RMI Registry identifying the relevant remote objects must be obtained (line 6), and the objects must be retrieved from the registry (lines 9 and 10),
3. A task must be created for each atomic sequence (lines 14, 35),
4. An Atomic Task Description must be created (lines 17 and 38), and the information about the upper bound on the number of calls to specific remote objects needs to be passed to it (lines 18, 19 for the first task and 39, 40 for the second),
5. The task body containing various operations, including remote method calls (line 25, 26, 27, 46, 47) must be defined, all surrounded by the methods representing the beginning (lines 22, 43) and end (lines 30, 53) of the task.

Similarly, an example of a basic server program, appropriate for the client program described above is shown in listing 2.3. The server is responsible for creating remote objects and placing them in the global registry.

The developer must ensure the following to successfully use Atomic RMI:

1. An RMI Registry must be obtained (lines 2, 3),
2. Remote objects are instantiated (lines 6, 7),
3. The objects are bound with a particular identifier (here: `"accountA"` and `"accountB"` – lines 10 and 11).

However, a situation may occur, where the definition of the remote object (such as that represented by the class `BankAccountImpl` in listing 2.3) can depend on remote objects defined elsewhere. For instance, one can imagine a virtual bank account, which, instead of containing actual physical assets, interfaces its two constituent accounts, and uses their resources whenever a withdrawal or a deposit is made. In case of such an account, whenever it is asked to perform an operation, the other two accounts may also need to also perform operations themselves.

In that case, the developer must take additional steps on the server-side to note such dependency. An example of such an application is presented in listing 2.4, and the additional necessary operations are as follows:

1. The composite object is initialized and given references to remote objects it requires (line 10),
2. A composition is defined with its dependencies, defining which objects the composition depends on, and how many calls to that object may be expected at most per each call to the composition (lines 13, 14, and 15),
3. The composition is bound with a particular identifier (here: `"accountAB"` – line 18).

To conclude, Atomic RMI facilitates development of distributed Java application by assuring serializability; however, Java RMI calls are regulated by Atomic RMI with the use of versioning algorithms, whose efficiency depends heavily on the whether the information about how many times, at most, each of the remote objects will be called is available. Therefore, the developer, though aided in providing serializability, is burdened by the task of analyzing his code and counting estimating maximum method call counts.

```
// Look up the central Atomic Task Manager.
AtomicTaskManager taskManager =
    AtomicTaskManagerService.lookupTaskManager("localhost");

// Obtain a Versioned RMI Registry.
Registry registry = VersionedLocateRegistry.getRegistry(taskManager);

// Fetch an account from the Registry.
BankAccountInf accountA = (BankAccountInf) registry.lookup("accountA");
BankAccountInf accountB = (BankAccountInf) registry.lookup("accountB");

/* The first concurrent task */ {
    // Initialize a new task.
    RmiAtomicTask task = new RmiAtomicTask(taskManager);

    // Create a preamble for the task.
    TaskDescription taskDescription = new TaskDescription();
    taskDescription.addObjectAccess(accountA, 2);
    taskDescription.addObjectAccess(accountB, 1);

    // Attach the preamble is to the task, and the task begins.
    task.startTask(taskDescription);

    // Transfer twice the amount from acocunt A to B.
    int a = accountA.withdraw(amount);
    a = a + accountA.withdraw(amount);
    accountB.deposit(a);

    // The task ends.
    task.endTask();
}

/* The second concurrent task */ {
    // Initialize a new task.
    RmiAtomicTask task = new RmiAtomicTask(taskManager);

    // Create a preamble for the task.
    TaskDescription taskDescription = new TaskDescription();
    taskDescription.addObjectAccess(accountA, 1);
    taskDescription.addObjectAccess(accountB, 1);

    // Attach the preamble is to the task, and the task begins.
    task.startTask(taskDescription);

    // Obtain information about balance on each account.
    int a = accountA.getBalance();
    int b = accountB.getBalance();

    // Display the information - an irrevocable I/O effect.
    System.out.println(a + b);

    // The task ends.
    task.endTask();
}
```

FIGURE 2.2: A simple client-side application of Atomic RMI.

```
// Obtain a Versioned RMI Registry.
Registry rmiRegistry = VersionedLocateRegistry
    .getRegistry(AtomicTaskManagerService.lookupTaskManager("localhost"));

// Initialize an object.
BankAccountInf accountA = new BankAccountImpl(1000);
BankAccountInf accountB = new BankAccountImpl(1500);

// Bind the object to an identifier.
rmiRegistry.rebind("accountA", accountA);
rmiRegistry.rebind("accountB", accountB);
```

FIGURE 2.3: A simple server-side application of Atomic RMI.

```
// Obtain a Versioned RMI Registry.
Registry rmiRegistry = VersionedLocateRegistry
    .getRegistry(AtomicTaskManagerService.lookupTaskManager("localhost"));

// Retrieve an object.
BankAccountInf accountA = (BankAccountInf)rmiRegistry.lookup("accountA");
BankAccountInf accountB = (BankAccountInf)rmiRegistry.lookup("accountB");

// Initialize a composite object, and pass object references to it.
BankAccountInf accountAB = new VirtualBankAccountImpl(accountA, accountB);

// Define the dependencies.
ObjectComposition composition = new ObjectComposition();
accountABcomposition.addDependency(accountA, 1);
accountABcomposition.addDependency(accountB, 1);

// Bind the composition to an identifier.
rmiRegistry.rebind("accountAB", accountAB, composition);
```

FIGURE 2.4: A server-side application of Atomic RMI featuring Compositions.

2.3 The Java precompilation tool for Atomic RMI

Atomic RMI would benefit from a tool to generate the source code that functions as preambles of tasks. Specifically, the library would gain in ease of use if the developer could avoid examination of the source code which is normally required in order to determine and enter the upper bounds on the number of method calls to all remote objects.

The Java precompilation tool consists of two main units: a unit responsible for static analysis and a unit for generating source code. The static analysis tool can be used to traverse the source code and provide an estimate on the number of invocations of methods requested from each identifiable remote object. The code generator can use the information obtained by static analysis and insert the appropriate source code into the program.

In the face of the fact that an exact number of method calls to an object, or an upper bound on that number, may not always be possible to evaluate using static code analysis, it is necessary for the tool to establish if the number has become unknown during the analysis and generate output conveying that information. For Atomic RMI using CVA, this means that the declaration of an upper bound on method calls for specific objects should be omitted from the output code in situations when the upper bound is unknown.

The Java precompilation tool is required to provide the necessary amendments to the source code before the compilation of the program. Therefore, it is obvious that the tool is required to create code that will not cause any compilation errors. Furthermore, it is necessary for the tool to be able to analyze the its generated code correctly and repeatedly.

The generated source code should be included into the original program, which, at that point, can be further modified by the developer. Therefore, it is essential for the tool to prevent obfuscation by limiting its changes to the original source code to only necessary modifications. It follows, that the original source code should contain all of its comments and its original indentation.

It is also required for the Java precompilation tool to work with Atomic RMI using the latter's own interface. Specifically, the generated code should feature calls to an object of type `TaskDescription` using the method `addObjectAccess` and two arguments: a variable pointing to a remote object, and the upper bound on the number of method calls to that object. an example of such code is line 16 of listing 2.2. It is assumed that these lines would or would not be present in the input file, and they would be generated correctly into the output file.

Similarly, the Java precompilation tool is demanded to generate code that calls instances of class `ObjectComposition` using the method `addDependency` and two arguments: a relevant variable pointing to a remote object, and the upper bound on the number of method calls to that object per one call to the aggregate object. An example of this code is in listing 2.4. Analogically to the client-side, it cannot be assumed whether these lines will be present int the input file, and it must be assured that they are generated in correct form into the output file.

Chapter 3

Algorithm

This chapter presents a simplified version of the algorithm that is used for the static code analysis of number of called methods in relation to objects: the first section briefly describes the notation used to express grammar in this chapter, the second section describes the language on which the algorithm operates, and the third section describes the method used for analysis: in particular, the functions that are used to handle individual constructs, the input and output of the algorithm. The final section presents a brief overview of a source code generator that uses the results of this algorithm.

3.1 Extended Backus-Naur Form

Extended Bachus-Naur Form[Int96] (EBNF) is a system of notation used to express context-free grammars; it is a superset of the popular Bachus-Naur Form[Con79] (BNF). This overview only concentrates on the features used further in the chapter.

EBNF defines production rules: each production is given a name followed by the equal sign and a comma-separated list of terminal symbols or other productions which is finalized by a semicolon. For instance,

```
production = 'terminal', non_terminal;
```

Terminal symbols are surrounded by single quotation marks. They represent symbols which cannot undergo any further subdivision, such as language keywords, variable names, etc.

Optionally, a number of alternatives to the list of symbols may be added: the various alternatives are separated with a vertical line character. For example, the following describes a digit, which can be any of the ten alternatives,

```
digit = '0' | '1' | '2' | '3' | '4'
      | '5' | '6' | '7' | '8' | '9';
```

Additionally, symbols may be surrounded with square brackets to suggest that they are optional, or with braces to suggest that the symbol may appear zero or more times at that position. For example, the following describes a number which can optionally be preceded by a minus, and which consists of one digit, and a series of zero or more other digits,

```
number = [ '-' ], digit , { digit };
```

3.2 The Simple Language

The language described here, called The Simple Language (SL), is analyzed by the algorithm described in 3.3. It was primarily devised to describe an extremely limited subset of constructs available in the Java language (although it can be used for other languages) which make up the most significant parts of the analysis, without cluttering the algorithm unnecessary detail.

The language consists of two categories of constructs: simple and composite expressions[1], and entities describing values and types.

The SL source can be expressed as an Abstract Syntax Tree, using the EBNF formalisms presented with each construct. The complete definition of SL is presented in Appendix A.

3.2.1 Entities

Entities are structures and ideas dealing with data values and their types.

Literal

A literal is a symbol defining a specific fixed value.

The predominantly important literals in the described language are Boolean values of *true* and *false*, which are represented in the same way in the Java language as:

```
true false
```

An EBNF notation for the literals is as follows:

```
literal = BOOLEAN;
BOOLEAN = 'true' | 'false';
```

Variable

A variable is a named entity of a specific type which holds a primitive value or points to an object.

A primitive value is a value which can be expressed in terms of literals. An object is an instance of a specific class, with a unique identity and a set of callable methods.

A variable is referred to in the code by its name, which is defined as follows:

```
variable = identifier;
identifier = ALPHABETIC_CHARACTER { ALPHANUMERIC_CHARACTER };
```

Here, the production *ALPHABETIC_CHARACTER* resolves to any lowercase and uppercase letters of the alphabet or the underscore; an *ALPHANUMERIC_CHARACTER* is either an *ALPHABETIC_CHARACTER* or a digit;

Type

A type is a description of an entity. A distinction can be made between primitive types and classes.

Primitive types describe primitive values, for instance, *true* and *false* are of type *Boolean.* Neither method calls nor initialization can be done on primitive type variables. There is a limited set of primitive types available in the language and no new primitive types may be added [Eck03].

Classes are complex types describing entities which may contain named methods. Thus, method calls may be done on variables that are of a class type. Additionally, classes have to be initialized — a class is only a description of behavior and structure, and becomes usable in the code through initialization, when it gains a state and a unique identity. An initialized class is called an object. New classes can be defined by the programmer [Som96].

```
type = identifier;
identifier = ALPHABETIC_CHARACTER, { ALPHANUMERIC_CHARACTER };
```

[1]In the terminology of region analysis these expressions map to *regions* — regions are defined as blocks of code with only one entry point [ALSU06], and both simple and composite expressions have single points of entry.

3.2.2 Simple expressions

Simple expressions are language constructs which act on individual variables, as opposed to entire expressions.

Variable declaration

A variable declaration is an expression which introduces a named variable and defines it in terms of type. It is a statement informing that the specific variable can hold values of that specific type.

As an example, a declaration of a variable called *o* of type *Object* can be declared in the Java language as,

```
Object o;
```

Formally, a variable declaration can be defined using EBNF as,

```
variable_declaration = type , variable;
```

Note that at the point of declaration the variable does not have any value — value can only be given through assignment (see 3.2.2).

Object initialization

An object initialization is an expression which creates a new instance of a given class. The initialization may be parametrized with zero or more arguments. The result of the initialization is that an object is created in the memory, it is given a specific type, and its constructor is executed.

In the Java language an initialization of a new object of type *Integer* with the literal *5* given as an argument would be represented as:

```
new Integer(5);
```

An EBNF definition of an object initialization can be expressed in the following manner,

```
object_initialization = type , { argument };
argument = variable | literal;
```

The object created through initialization is not identified by any name at this point. This is why it is usual to immediately assign the result of this operation to a variable (see 3.2.2).

Variable assignment

A variable assignment expression defines that a variable changes its current value to the value dictated by the given expression. The precise behavior of the assignment is established as follows:

- If the expression is a literal, a variable holding a primitive value or an expression evaluating to a primitive value, then the variable will take the value of the literal;
- If the expression is a variable pointing to an object, an object initialization, or an expression evaluating to an object, then the variable will point to that object.

In the Java language, an assignment of the Boolean literal *true* to a variable named *b* would be done as follows:

```
b = true;
```

Similarly, a Java example of an assignment of a newly initialized object of type *Object* to a variable named *o*, and then pointing another variable *p* to that same object, would be:

```
o = new Object();
p = o;
```

An EBNF representation of a variable assignment can be presented as follows,

```
variable_assignment = variable , expression
                    | variable , literal
                    | variable , variable;
```

Method call

A method call is an expression used to indicate that an object will be asked to execute a method. The call may be parametrized with zero or more arguments.

In the Java language, a request to an object *o* to call a method *method* with the literals *5* and *true* as arguments would be represented as:

```
o.method(5,true);
```

A method call can be expressed using EBNF notation in the following manner,

```
method_call = object , method;
method = identifier , { argument };
argument = variable | literal;
```

To simplify the explanation of the algorithm in 3.3, the methods are assumed to be pure. This means that they are assumed not to modify any objects outside of whatever objects are created inside the method.

3.2.3 Composite expressions

Composite expressions are expressions which act on, or consist of other expressions.

Block

A block is an expression which itself contains a list of expressions. The list may contain any number of expressions, including none.

A block defines a scope of the variables that are declared within it; that is, variables from outside the scope can be accessed and modified without hindrance from within the scope. On the other hand, the variables that are declared within the scope cannot be accessed or modified outside of it — they can only be used inside the block, or inside any blocks that are nested therein.

In the Java language, a block would be denoted with a surrounding pair of braces. For example, a block containing two method calls would be written as

```
{
    a.call();
    b.call();
}
```

A block can be expressed as EBNF in the following way,

```
block = { expression };
```

Conditional block

A conditional block is an expression composed of an expression acting as a condition, and a positive block which is a block of code that will be executed if the condition evaluates to *true*. Optionally a conditional block may include an alternative block, which will be executed if the condition evaluates to *false*.

In the Java language, a conditional expression can be expressed with the keyword *if* followed by a conditional expression (surrounded by parentheses) and a block of code. An alternative block of code may be expressed appending the keyword *else* and a block of code to that construct.

An example of the conditional expression in the Java language, which, depending on whether the method *isRunning* returns *true* or *false*, will either call the method *stop* or *start* (respectively) on the variable *a*.

```
if (a.isRunning()) {
    a.stop();
} else {
    a.start();
}
```

The conditional expression can be expressed using EBNF as follows,

```
conditional_block = condition, positive_block,
                    [ alternative_block ];
condition = expression | literal;
positive_block = block;
alternative_block = block;
```

Loop block

A loop block is an expression composed of a condition and a body, which is a block of code that is executed as long as the conditional expression evaluates to *true*. This means that the block of code may be executed zero or more times.

In the Java language, the most basic loop may be expressed as,

```
while (a.isRunning()) {
    a.stop();
}
```

An EBNF representation of a loop can be expressed as follows:

```
loop_block = condition, body;
condition = expression | literal;
body = block;
```

3.3 Method Call Count Analysis

The Method Call Count Analysis (MCCA) algorithm is a method of statically analyzing a piece of source code expressed in SL. The goal of MCCA is to count the upper bound on the number of calls requested from a predefined set of objects.

3.3.1 Safety

Due to the fact that the analysis operates on approximations to the actual execution of programs, an analyzer is obliged to assure that it follows a *safety*-preserving policy in cases where these approximations could result in erroneous output[ALSU06]. Specifically to the problem at hand, the property of safety can be said to be satisfied if the analysis cannot produce any upper bound on the number of method calls that is lower than an actual number of method calls during any execution of a program.

Thus, MCCA satisfies safety for programs written in SL by defining the join operator (presented in equation 3.20) and the arithmetic governing the operations of addition, multiplication, and maximum (equations: 3.2, 3.3, and 3.4, respectively) in such a way that, in an event of uncertainty, the safer solution will always be chosen.

For example: in a case when the number of method calls may be dependent on a condition, whose outcome cannot be evaluated statically, the analysis will select the safer among the possible number — the greater number. If, on the other hand, the upper bound must be specified from a set of possibilities including an element whose value is unknown or uncertain, then the resultant upper bound will also be uncertain.

3.3.2 Input

SL in the form of an expression tree should be provided as an input for the algorithm. The expression tree is generated from the source code of the program following the description in 3.2.

Figure 3.1 presents an example program that complies with SL,

```
{
    Thread a;
    a = new Thread();
    a.start();

    while (a.isRunning()) {
        a.getProgress();
    }

    boolean b;
    b = true;
    if (a.isSuccessful()) {
        b = false;
    } else {
        a.restart();
    }
}
```

FIGURE 3.1: An example program in SL.

The following tree can be constructed using the listing:

3.3.3 Output

The output of the algorithm is a map whose keys are unique identifiers of objects. The particular method of generating these identifiers is not important from the algorithms point of view, as long as they provide a way to distinguish among the objects. The variable names will not normally be used in this function, since a single variable may point to many objects during the course of a single program. [2]

The keys are mapped to values represent the upper bound on the number of times each of the objects were used to make a method call in the program. This number can be zero, any finite natural number, or ω if the number of method calls to an object cannot be predicted by static analysis.

[2] In the actual implementation, only remote objects are considered, so their identifiers are generated using their URIs.

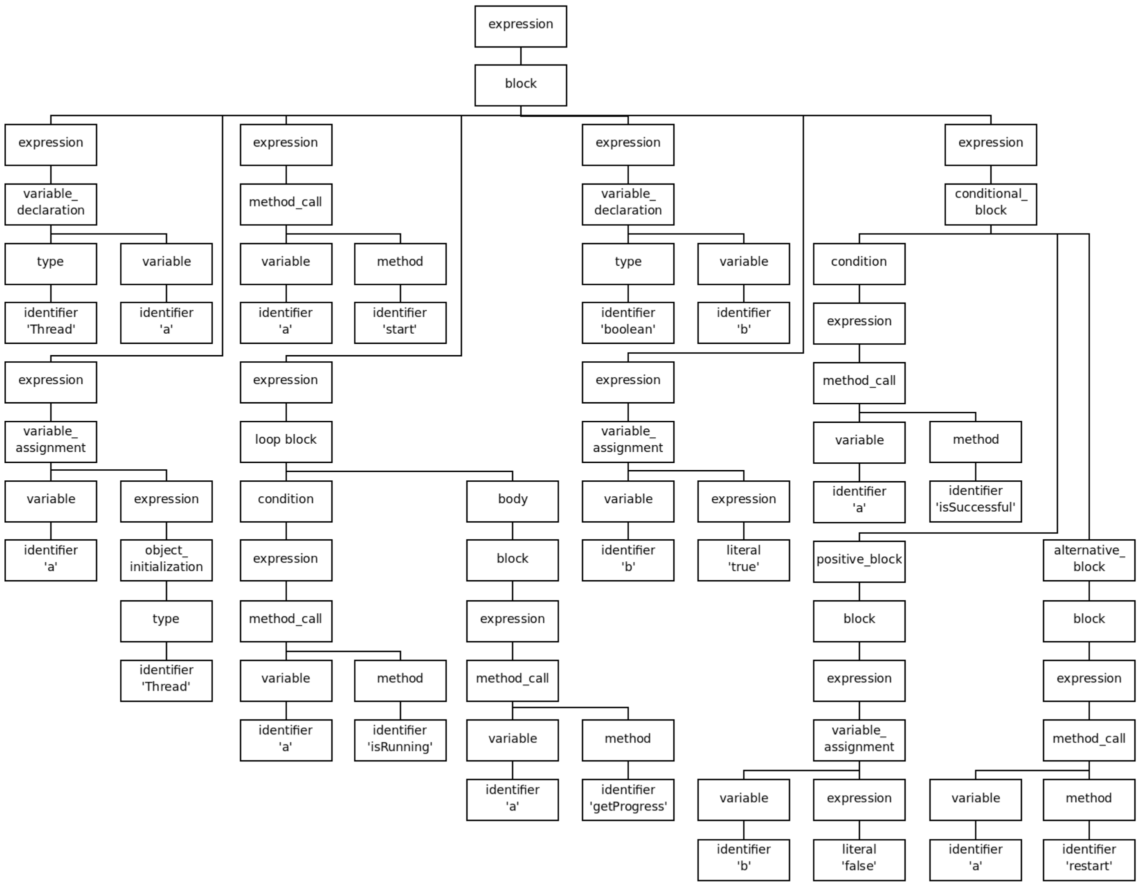

FIGURE 3.2: An example AST of a program.

Formally, the output map may be defined as follows:

$$Output : \{identifier \rightarrow \mathbb{N}_0 \cup \omega\} \tag{3.1}$$

where, $identifier$ belongs to the set of all possible identifiers.

During the course of the analysis, operations will be performed on the values from the map: the values will be added, multiplied, and compared. It is, therefore, essential to explain the behavior of ω in such situations.

Addition

The result of addition equals ω if any of the terms equal ω. That is to say, if any number is added to ω, the resulting value will equal ω; similarly, if ω is added to itself, the resulting value will also equal ω.

The simple rationale behind that definition is that, if a number of method calls for a specific objects is unpredictable, no matter what other value is added to that number, it remains unpredictable. This assumption helps ensure the property of safety for the MCCA, as described in section 3.3.1.

Addition of numbers, when one of the terms is equal to ω can be formalized as follows,

$$\omega + x = \omega \qquad \text{where} x \in \mathbb{N}_0 \cup \omega \tag{3.2}$$

Multiplication

The result of multiplication equals ω if any of the factors equal ω. That is to say, if any number is multiplied by ω, the resulting value will equal ω; similarly, if ω is multiplied by itself, the resulting value is ω.

As with addition, the rationale is, that if any number is multiplied by an unpredictable value, then the outcome is also unpredictable, with the sole exception of when one of the factors has the value of 0, in which case the outcome will equal 0. This allows to maintain the safety of the analysis, as defined in section 3.3.1.

Multiplication of numbers, when one of the factors is equal to ω can be formalized as follows,

$$\begin{aligned} &\omega * 0 = 0 \\ &\omega * x = \omega \qquad \text{where} x \in \mathbb{N}_0 \cup \omega \end{aligned} \tag{3.3}$$

Maximum

The function max returns ω, if at least one of its parameters is ω

Since it cannot be predicted whether any other arguments of the function max will be greater or equal to ω, it cannot be predicted which particular number will be chosen as the result. So, the result is unpredictable. As a result of such a definition, the maximum operation aides the enforcement of safety, as specified in section 3.3.1.

The choice of the highest number can be formally expressed as,

$$\max(x_0, x_1, \dots, x_n) = \omega, \qquad if \exists k,\ x_k = \omega \tag{3.4}$$

3.3.4 Method

The algorithm is started by the application of function **analyze_expression** (discussed below) to the root of the provided expression tree. It is assumed that entire program is executed only once, so the number of iterations provided as the second argument to the function is 1. Additionally, no information about method calls are available at the start of the algorithm.

Expression analysis

An initial analysis of each expression is done by function **analyze_expression**, which takes the expression tree *(*E), provided as an argument, decides what particular type of expression it represents, and then delegates the analysis to an appropriate specialized function. The number of times that the evaluation of the expression takes place (argument n) is passed to the specialized functions without modification. And the function returns the output of the specialized function.

Formally,

$$\begin{aligned} &analyze\_expression(E, n) = \\ &\begin{cases} analyze\_declaration(child(E), n) & \text{if } type(E) = \text{variable\_declaration} \\ analyze\_initialization(child(E), n) & \text{if } type(E) = \text{object\_initialization} \\ analyze\_assignment(child(E), n) & \text{if } type(E) = \text{variable\_assignment} \\ analyze\_call(child(E), n) & \text{if } type(E) = \text{method\_call} \\ analyze\_block(child(E), n) & \text{if } type(E) = \text{block} \\ analyze\_conditional\_block(child(E), n) & \text{if } type(E) = \text{conditional\_block} \\ analyze\_loop\_block(child(E), n) & \text{if } type(E) = \text{loop\_block} \end{cases} \end{aligned} \tag{3.5}$$

The type of the function is defined as follows:

$$analyze\_expression : expression \rightarrow (\mathbb{N}_0 \cup \omega) \rightarrow \{identifier \rightarrow \mathbb{N}_0 \cup \omega\} \tag{3.6}$$

Function **type** is used to determine the type of the expression. It simply returns one of *variable_declaration*, *object_initialization*, *variable_assignment*, *method_call*, *block*, *conditional_block*, *loop_block*, depending on what type the first child of the root of the expression subtree actually indicates.

Variable declaration, object initialization, and variable assignment analysis

Functions **analyze_declaration** and **analyze_initialization** are presented here in a very simplified form, as they have only an indirect relation to the analysis responsible for counting method calls.

The functions take two arguments,

- E – an expression tree representing a variable declaration, or object initialization,
- n – an upper bound on the number of times this expression will be evaluated,

and return a map defined as *Output* (equation 3.1). Therefore, the functions may be defined in terms of type in the following fashion:

$$analyze\_declaration : variable\_declaration \rightarrow (\mathbb{N}_0 \cup \omega) \rightarrow \{identifier \rightarrow \mathbb{N}_0 \cup \omega\} \tag{3.7}$$
$$analyze\_initialization : object\_initialization \rightarrow (\mathbb{N}_0 \cup \omega) \rightarrow \{identifier \rightarrow \mathbb{N}_0 \cup \omega\} \tag{3.8}$$

Since the expressions analyzed here are simple expressions (they do not have other expressions nested inside), they cannot contain method calls at all, therefore both functions return an empty map. Or, formally:

$$analyze\_declaration(E, n) = \emptyset \tag{3.9}$$
$$analyze\_initialization(E, n) = \emptyset \tag{3.10}$$

The contents of the expressions mentioned here are important to other analyses, which should be run alongside the method call analysis to keep track of values of variables, object pointers, etc. These analyzes have been omitted in this descriptions for reasons of clarity.

Assignment analysis

Function **analyze_assignment** takes two arguments,

- E – an expression tree representing a method call, which contains a subtree representing the expression, literal or variable, which is assigned to a variable;
- n – an upper bound on the number of times this expression will be evaluated.

and returns a map defined as *Output* (equation 3.1). Therefore, the function may be defined in terms of type as the following:

$$analyze\_assignment : variable\_assignment \rightarrow (\mathbb{N}_0 \cup \omega) \rightarrow \{identifier \rightarrow \mathbb{N}_0 \cup \omega\} \tag{3.11}$$

Function **analyze_assignment** is technically a simple expression, but it can be logically linked with an expression that returns a value. Therefore the analysis of the assigned expression is conducted (delegated to **analyze_expression**) in case of an actual statement being assigned. Otherwise, the expression is treated like other simple expressions that do not call methods, and an empty map is returned.

Formally, the function can be expressed as

$$\begin{aligned}
&analyze\_assignment(E, n) = \\
&\quad\begin{cases} analyze\_assignment(V, n) & \text{if } is\_expression(V) \\ \emptyset & \text{if } \neg is\_expression(V) \end{cases} \\
&where \\
&\quad V = assigned(E)
\end{aligned} \tag{3.12}$$

Function **assigned** returns a subtree representing the assigned expression, variable, or literal. Function **is_expression** checks whether a given subtree represents an expression.

The analysis of assignments is otherwise of importance to auxiliary analyzes, which were not described here to maintain simplicity.

Method call analysis

Function **analyze_call** takes two arguments,

- E – an expression tree representing a method call, from which can be discerned the names of the variable, and the method;
- n – an upper bound on the number of times this expression will be evaluated.

and returns a map defined as *Output* (equation 3.1). Therefore, the function may be defined in terms of type as:

$$analyze\_call : method\_call \rightarrow (\mathbb{N}_0 \cup \omega) \rightarrow \{identifier \rightarrow \mathbb{N}_0 \cup \omega\} \tag{3.13}$$

The function returns a map containing a single element, which maps the subject of the analyzed call, to a number of times this expression will be evaluated (the number of times this method call will be made).

This can be expressed formally as,

$$analyze\_call(E, n) = . \ \{object(v) \mapsto n\}, \quad v = variable(E) \tag{3.14}$$

Function **variable** extracts from the tree the identifier of the variable that the method is called on. Function **object** establishes which object is pointed to by the variable with a specified identifier.

For reasons of clarity, all methods are considered pure: the methods' executions do not affect variables that are not defined within the body of the method. That is, the methods do not modify the states of their arguments or the states of any other objects defined outside of them. This means, that, from the point of view of the algorithm the analysis of their contents can be overlooked, as no major changes to the objects can come from within those methods.

Were the methods impure, an additional analysis would be required of their bodies of these methods, in order to establish what effect their execution has on other variables, and to count calls to methods of remote objects within the bodies. Additionally, a possibility of recursion would require consideration, whose outcome cannot be reliably predicted by static analysis in all cases.

Block analysis

Function **analyze_block** takes two arguments,

- E – an expression tree representing a block, that contains zero or more expressions as its children;
- n – an upper bound on the number of times this block will be evaluated.

and returns a map defined as *Output* (equation 3.1). Therefore, the function may be defined in terms of type int this fashion:

$$analyze\_block : block \rightarrow (\mathbb{N}_0 \cup \omega) \rightarrow \{identifier \rightarrow \mathbb{N}_0 \cup \omega\} \tag{3.15}$$

The function takes all of the expressions contained within the block and calls function **analyze_expression** on them, passing the upper bound on the number of evaluations without modification.

The result of the function is created by folding the results of function **analyze_expression** for all contained expressions.

The result map R is initially empty. Folding proceeds by using the function **fold** for each map M returned by **analyze_expression** run for each expression contained in the block. This can be expressed formally as the following:

$$\begin{aligned} &analyze\_block(E, n) = \\ &\qquad add(R, analyze\_expression(E_{child}, n)), \forall E_{child} \in children(E) \end{aligned} \tag{3.16}$$

Function **children** returns all children of the root node of a given tree.

Function **add** adds the contents of one map to the contents of another map; that is, for each key k in map M, which is not a key in result map R, the key-value pair is copied from M to R. For each key k which is present in both M ad R, the new value it maps to in R is equal to the sum of both values from maps M and R. Or, formally,

$$add(R, M) = \begin{cases} R[k] = M[k], & \forall k \in keys(M), k \notin keys(R) \\ R[k] = M[k] + R[k], & \forall k \in keys(M), k \in keys(R) \end{cases} \tag{3.17}$$

where M and R are maps, mapping identifiers to upper bounds on method call counts. Thus, the type of function **add** can be expressed in the following terms:

$$add : \{identifier \rightarrow \mathbb{N}_0 \cup \omega\} \rightarrow \{identifier \rightarrow \mathbb{N}_0 \cup \omega\} \rightarrow \{identifier \rightarrow \mathbb{N}_0 \cup \omega\} \tag{3.18}$$

Function **keys** returns a set containing all the keys present in a given map.

Conditional block analysis

Function **analyze_conditional_block** takes two arguments,

- E – an expression tree representing a conditional block, that contains a condition and one or two block expressions;
- n – an upper bound on the number of times the conditional block will be evaluated.

and returns a map defined as *Output* (equation 3.1). Therefore, the function may be defined in terms of type as the following:

$$analyze\_conditional\_block : conditional\_block \rightarrow (\mathbb{N}_0 \cup \omega) \rightarrow \{identifier \rightarrow \mathbb{N}_0 \cup \omega\} \tag{3.19}$$

The function attempts to evaluate the value of the condition. The condition can be either a literal or an expression.

In the case of a literal condition, the possible values are *true* or *false*. In case of *true*, the analysis progresses by calling **analyze_expression** using the positive block as the first argument, and passing the upper bound on the number of executions without change; the function then returns the results of **analyze_expression**. In the case of *false* and if there is an alternative block present, the execution proceeds as above, with the sole exception that the first argument is the alternative block and not the positive block. If there is no alternative block, no further action is taken and the function returns an empty map.

On the other hand, if the condition is an expression, it has first to undergo a preliminary evaluation and an analysis. The analysis is conducted by calling **analyze_expression** using the conditional expression as the first argument, and passing the upper bound on the number of executions without change.

The process of preliminary evaluation is a separate analysis (not described here) and may indicate that the conditional expression evaluates to one of the three possible values: *true*, *false*, or *unpredictable*. In the case of *true* and *false* the same procedure is followed as with literal conditions, except that the results of **analyze_expression** are added to the method call results of the analysis of the condition, and the result of that operation is returned. In the case of *unpredictable*, both blocks are evaluated using **analyze_expression**, but their results are joined.

The symbol $\bigvee$ represents the join operator. Join takes two maps M and N, and returns a new map R, which contains the upper bounds for all values contained in M and N; that is, for all keys which are in N but not in M, the key-value pairs are copied to the R. Likewise, for all keys which are in M but not in R, the key-value pair are also copied to R. Finally, for each key k which is both in M and N an entry is inserted into R, where the key is k and the value is equal to the maximum of the values that k maps to in M and N. This can be expressed as,

$$\begin{aligned} &M \bigvee N = R, \\ &\quad R[k] = M[k], && \forall k \in keys(M) \setminus keys(N) \\ &\quad R[k] = N[k], && \forall k \in keys(N) \setminus keys(M) \\ &\quad R[k] = \max(M[k], N[k]), && \forall k \in keys(N) \cap keys(M) \end{aligned} \tag{3.20}$$

The join operator can be expressed in terms of type in as the following:

$$\bigvee : \{identifier \to \mathbb{N}_0 \cup \omega\} \to \{identifier \to \mathbb{N}_0 \cup \omega\} \to \{identifier \to \mathbb{N}_0 \cup \omega\} \tag{3.21}$$

The entire analysis can be expressed with the following formalism,

$$\begin{aligned} &analyze\_conditional\_block(E, n) = \\ &\begin{cases} analysis\_proper(E, C, n) & \text{if } is\_literal(C) \\ add(analyze\_expression(C, n), analysis\_proper(E, C, n)) & \text{if } \neg is\_literal(C) \end{cases} \\ &where \\ &\quad C = condition(E) \end{aligned} \tag{3.22}$$

Function **condition** returns a subtree representing the condition of a conditional block. Function **is_literal** checks whether the provided tree representation is a literal.

Function **analysis_proper** is responsible for the analyzing the proper block, once the condition itself was handled accordingly. It is defined as,

$$analysis\_proper(E, C, n) = \begin{cases} analyze\_expression(E_{positive}, n) & \text{if } is\_true(C) \\ analyze\_expression(E_{alternative}, n) & \text{if } is\_false(C) \\ \\ analyze\_expression(E_{positive}, n) & \\ \bigvee analyze\_expression(E_{alternative}, n) & \text{if } \neg is\_true(C) \wedge \neg is\_false(C) \end{cases} \tag{3.23}$$

$$\begin{aligned} &where \\ &\quad E_{positive} = positive\_block(E), \\ &\quad E_{alternative} = alternative\_block(E) \end{aligned}$$

The type of function **analysis_proper** is presented in the following equation:

$$analysis\_proper : conditional\_block \rightarrow condition \rightarrow (\mathbb{N}_0 \cup \omega) \rightarrow \{identifier \rightarrow \mathbb{N}_0 \cup \omega\} \tag{3.24}$$

Function **positive_block** returns a subtree representing the positive block of a conditional block. Function **alternative_block** returns a subtree representing the alternative block of a conditional block, or an empty block if no alternative block has been defined. Functions **is_true** and **is_false** checks whether the provided condition evaluates to *true* or *false* (respectively) no matter whether the argument is a literal or an expression.

The join operator is used in cases when the condition may evaluate to either *true* or *false* during the actual evaluation of the program. It ensures that the maximum number of method calls will be selected as a result of the analysis, thus satisfying the condition of safety, as defined in section 3.3.1.

Loop block analysis

Function **analyze_conditional_block** takes two arguments,

- E – an expression tree representing a loop block which contains a condition and a block of expressions;
- n – an upper bound on the number of times the conditional block will be evaluated.

and returns a map defined as *Output* (equation 3.1). Therefore, the type of the function can be presented, as follows:

$$analyze\_loop\_block : loop\_block \rightarrow (\mathbb{N}_0 \cup \omega) \rightarrow \{identifier \rightarrow \mathbb{N}_0 \cup \omega\} \tag{3.25}$$

The function attempts to analyze an expression representing a loop. In general, it is assumed that the number of evaluations of a loop is unknown, and an attempt at establishing the fact can be done. If the attempt succeeds, this value is assigned to m, or if the attempt fails, or is not made at all, m becomes ω, that is, it is assumed that the number of evaluations is unpredictable. When m is defined, further analysis of the body is delegated to the function **analyze_expression**, with the body of the loop as the first argument, and the result of the multiplication of m and n as the second. Additionally, if the condition of the loop is an expression, instead of a literal value, an analysis of the condition is preformed using **analyze_expression**, before the analysis of the body.

$$
\begin{aligned}
&\mathit{analyze\_loop\_block}(E, n) = \\
&\quad\begin{cases} \mathit{analyze\_expression}(E_{body}, n * m) & \text{if } \mathit{literal}(C) \\ \mathit{add}(\mathit{analyze\_expression}(C, n * m), \mathit{analyze\_expression}(E_{body}, n * m)) & \text{if } \neg\mathit{literal}(C) \end{cases} \\
&\mathit{where} \\
&\quad C = \mathit{condition}(E), \\
&\quad E_{body} = \mathit{body}(E), \\
&\quad m = \mathit{evaluation\_count}(E)
\end{aligned}
\tag{3.26}
$$

Function **evaluation_count** analyzes the loop block and returns the upper bound on the number of times the loop will be executed when the program is run. The values returned by the function be in the same domain as method call, i.e., any integer greater or equal to zero, or the symbol ω representing an unpredictable number. The type of the function is,

$$\mathit{evaluation\_count} : \mathit{loop\_block} \rightarrow (\mathbb{N}_0 \cup \omega) \tag{3.27}$$

The current implementation of the **evaluation_count** function is simplified: in its current state the function returns 0, if it can be predicted that the loop will not be evaluated, and ω in any other case. Such a definition ensures that the number of evaluations is not underestimated. The function is formally defined by the following equation:

$$
\begin{aligned}
&\mathit{evaluation\_count}(E) = \\
&\quad\begin{cases} 0 & \text{if } \mathit{is\_false}(\mathit{condition}(E)) \\ \omega & \text{if } \neg\mathit{is\_false}(\mathit{condition}(E)) \end{cases}
\end{aligned}
\tag{3.28}
$$

Even though the problem of counting the exact number of evaluations of a loop is not solvable by means of static analysis, it is certainly possible to arrive at a more precise algorithm than the function presented here, for instance, by the use of symbolic analysis of the loop. However, such an approach has not been implemented.

Additionally — though it is a key to performing the analysis of loops — if **evaluation_count** is not defined, a constant function returning ω in all cases can be used in its stead without causing the number of calls to be underestimated. Alas, this will eventuate in the analysis of all method calls within the loop providing very imprecise results.

3.4 Code generation

The MCCA algorithm has an application in code generation for libraries that require the knowledge about the upper bound on the number of method calls done on individual objects. One such library is Atomic RMI [Mam08] whose versioning algorithm's performance is improved if the upper bounds on method calls are known in advance.

3.4.1 Input

The code generator takes source code as input. Particularly the source code contains a description of variables that are relevant to the versioning algorithm, and a number of definitions of *tasks*, each consisting of a *header*, where the task meta-data is placed, and a *body*, which defines the operations that the task performs.

The following listing presents an example source code, for the Atomic RMI library, with comments pointing out the aforementioned elements.

```
// Initialization of a system-wide task manager.
AtomicTaskManager taskManager =
        AtomicTaskManagerService.lookupTaskManager("example.com");

// Declaration of remote objects.
Registry registry = VersionedLocateRegistry.getRegistry(taskManager);

RemoteObject a = (BankAccountInf) registry.lookup("remoteObjectA");
RemoteObject b = (BankAccountInf) registry.lookup("remoteObjectB");

// The header of the task.
RmiAtomicTask task = new RmiAtomicTask(taskManager);
TaskDescription taskDescription = new TaskDescription();

// The body of the task.
task.startTask(taskDescription);

a.call();
if (random()) {
    b.call();
}

task.endTask{taskDescription};
```

3.4.2 Output

As output, the code generator reproduces the source code received as input, but with additional code in each task header describing the upper bound on the number of method calls done on each of the objects relevant objects. In case the upper bound is unknown for a given object, the generated code only contains a declaration that the object will be accessed, but the exact number of calls is omitted.

For instance, a task header generated from the code in the listing in subsection 3.4.1 would be generated as presented in figure 3.3.

```
// The header of the task.
RmiAtomicTask task = new RmiAtomicTask(taskManager);
TaskDescription taskDescription = new TaskDescription();

taskDescription.addObjectAccess(a, 1);
taskDescription.addObjectAccess(b, 1);
```

FIGURE 3.3: An example of a generated Atomic RMI task header.

3.4.3 Method

Code generation proceeds by analyzing the source code in search of objects relevant to the generator. Each for these objects is given a unique identifier, that corresponds to the unique identifier given to that object by MCCA.

Next, the code generator finds all tasks defined in the code. For each task, MCCA is run, producing a result set that maps objects' unique identifiers to the upper bound on the number of method calls to those objects. It is essential that the generator produces an insert for all objects that are referenced in the task.

For each object's unique identifier in the result set for a given task, a line of code is inserted into the original source, with a variable name that points to that object, and an upper bound on method calls, unless the upper bound is unpredictable or the object is not referenced by any variable accessible within the task's header.

Additionally, if an upper bound on the number of method calls for any variable is defined manually in a task's header, the generator preserves the manual definition and refrains from generating a line of code for that object in that task.

Chapter 4

Tools

This chapter describes and compares the various parser generators which can be used to develop a Java precompiler.

The first section constitutes an explanation of specialist terms appearing in this chapter. The following section provides an overview for each tool, describing briefly the major available compiler compilers: CUP/JFlex, JavaCC, SableCC, and ANTLR. The final section presents the rationale for ANTLR to be a satisfactory and appropriate basis for the Java precompiler.

4.1 Introduction to parsing and static code analysis

The description of tools for static analysis requires a few terms to be explained. This section presents a set of terms that appear in the current, and the following chapters. The descriptions follow [ALSU06].

4.1.1 The process of static code analysis

Static code analysis is an analysis of the source code of a program before it is executed (in contrast to *dynamic code analysis*). Typically the process of analyzing source code consists of *lexical*, *syntactic*, and *semantic* analyses.

Lexical analysis is performed by a program called a *scanner*, *lexical analyzer*, or *lexer*. The function of a scanner is to convert the individual characters of the source code into a stream of *tokens* – a descriptive symbol related to a particular meaningful sequence of characters.

Syntactic analysis is performed by a *parser*, otherwise called a *syntax analyzer*, whose function it is to convert a token stream into an intermediate representation, such as an *Abstract Syntax Tree* (AST). An AST is a tree-like structure where each node represents an operation, and its children represent the arguments of that operation.

Semantic analysis can take a variety of forms (e.g. type checking) and it is concerned with gathering information from an AST to be used in a further phase of the process, such as intermediate code generation.

4.1.2 Parser generation

Manual construction of parsers and lexers is rare; indeed, it is common practice to create such programs using appropriate *generators*. These generators take a language *grammar* (with some additional information) and automatically produce programs capable of analyzing the language described by that grammar.

A grammar is a specification for a formal language, that consists of a set of rules, called *productions*, that describe, broadly, how any symbol can be mapped into a list of other symbols. Grammars are described using a formal notation that is typically derived from *BNF*, or *EBNF*, the latter of which is presented in brief in Section 3.1.

Generated parsers fall into one of several parser classes, where each class describes such elements as the number of symbols that are read from input (*lookahead*) and the direction of parsing. Popular classes of parsers include *LALR(1)*, and *LL(k)/LL(*)* parsers.

LALR(1) is a bottom-up parser that parses input from left to right with a single token of lookahead. LALR(1) parsers are capable of analyzing a larger set of languages than LL(k) parsers, and they are considered to be highly efficient.

LL(k) is a top-down parser that parses input from left to right and constructs a leftmost derivation; it uses k tokens of lookahead during parsing. LL(k) parsers can operate on a smaller number of languages than LALR(1) without backtracking, however they are generally more intuitive. LL(*) parsers are a variant that allow an arbitrary lookahead, instead of a fixed, finite lookahead value [Par07].

4.2 Overview of parser generators

A number of high-quality parser generators have been created, each tool with its own unique set of characteristic features, both advantageous and otherwise. This overview means to provide a brief description of each of these tools.

4.2.1 CUP/JFlex

CUP[Hud06] (Constructor of Useful Parsers) is a parser generator for the Java language, designed for use with simple grammar specifications. It has been developed by Scott E. Hudson at the Georgia Institute of Technology, and was heavily inspired by YACC [Joh75](Yet Another Compiler Compiler), which was written for the C language. Currently, CUP is maintained by the University of Munich.

CUP generates LALR(1) one-pass parsers from a grammar specification that bears resemblance to BNF. The grammar specifies parser actions — programmatic activities to be executed when a particular production is evaluated — which generate output or perform other operations.

Like YACC, CUP does not generate lexical analyzers by itself, but instead interfaces with dedicated scanner generators. JFlex[Kle09] is the most common lexical analyzer used with CUP. It is a complete Java re-write of Lex[Les75], which originally was concocted to produce C code, and which was the scanner generator used with YACC. JFlex was created by Gerwin Klein.

Among the advantages of CUP/JFlex there are superior performance and the ability to analyze left-recursive grammars. The most serious of the disadvantages is the lack of support for AST generation; others include: bloat of the specification caused by embedding action code within the grammar, lack of UTF-8 support, and occasional problematic cooperation between the generated parser and scanner.

4.2.2 JavaCC

JavaCC[Sun03a] (Java Compiler Compiler) is a parser and lexical analyzer generator for Java. JavaCC was originally implemented by Sreeni Viswanadha and Sriram Sankar for Sun Microsystems.

JavaCC generates LL(k) parsers (with variable length lookahead capabilities) from a grammar specification that takes the form of a set of functions responsible for parsing individual productions. The functions use EBNF-like constructs, like closures and optional clauses, to allow greater flexibility in describing the productions.

JavaCC was originally intended to only support action code (instructions that are evaluated on emergence of a certain type of production). However, there are two JavaCC-based tree building tools available that allow creation of ASTs: JTB[UCL05] (Java Tree Builder) and JJTree[Sun03b] (developed alongside JavaCC). Therefore, JavaCC can be used to create multiple-pass parsers.

The most significant advantage to JavaCC is the ability to create ASTs automatically; other advantages are UTF support and the support for recursive-descent tree parsers, which are generally considered more intuitive, support for tree pruning, and, finally, the ability to set the lookahead value dynamically. The disadvantages include lack of general ability to process left-recursion in the grammar specification (which, nonetheless, can be partially remedied by increasing the lookahead), and performance lower than LALR(1) parser generators.

4.2.3 SableCC

SableCC[Gag98] (Sable Compiler Compiler) is a scanner, parser and tree-walker generator for Java, conceived by Étienne Gagnon and implemented at the McGill University, Montreal.

SableCC generates LALR(1) parsers using EBNF-like grammar on input. The resultant parser produces strongly-typed ASTs as its output. In addition to the parser (and scanner) a tree-walker framework for the grammar is automatically generated, to facilitate the implementation of AST analyses.

The advantages to SableCC are the generation of ASTs in a way that enforces separation of grammar specification and analyzer code, and improved performance over LL(k) solutions. The disadvantages, are time- and memory-consuming code generation, unwieldiness, and unintelligibility of the analyzer source code, rendered thus by the excessive amount of type casting, forced upon the programmer by the use of strongly-typed, inflexible, unpruned ASTs.

4.2.4 ANTLR

ANTLR[Par07] (ANother Tool for Language Recognition) is a scanner and parser generator for the Java language, created by Terrence Parr at the University of San Francisco, constructed on a similar earlier such program of his – PCCTS[Par97] (Purdue Compiler Construction Tool Set).

ANTLR generates LL(*) parsers using grammar specifications defined in an EBNF-like language. Depending on the grammar, an AST can be generated, or parser actions can be performed.

A characteristic feature of ANTLR are semantic predicates – mechanisms that allow the parser to avoid ambiguous, context-sensitive constructs, by providing additional conditions that are evaluated during runtime. Using these conditions, productions, or parts of productions, may be excluded from the analysis at any given time.

The advantages of ANTLR include the ability to produce specifications that separate the generation of ASTs and the implementation of the analyzer, UTF support for the grammar specification, creation of intuitive top-down (recursive-descent) parsers, support for pruning, a dynamically modifiable lookahead value, and a backtracking mechanism. Additionally, the use of semantic predicates aides the creation of special-purpose parsers, where semantic analysis will closely follow semantic analysis. ANTLR is disadvantaged by potential lower performance and lower legibility of the grammar when semantic predicates are used.

4.3 A parser generator for the Java precompiler

For the purpose of selecting a parser and lexer generator acting as the backend for the Java precompiler tool, the following specific features are desirable.

First of all, the precompiler necessitates multiple analyses to be performed, during which modifications to the analyzed structure may be required. Hence, this particular application requires the parser to create a flexible AST.

The existence of liberally-licensed grammar specifications for a current version of the Java language is a necessity, since the precompiler requires a parser for the Java language, but, taking into account the available resources and deadlines, the effort required to produce a complete specification may prove to be too great.

Furthermore, it is required that the tool generates source code using the original source files as a template. The generated code should not become bereft of the programmer's comments, nor should the original indentation and general whitespace arrangement become mangled. On the other hand, comments and whitespaces should be excluded from the analysis itself, not to dim the view.

From the presented parser and lexer generators, ANTLR proves to provide all of the aforementioned characteristics. First of all, not only is AST generation supported, but it is additionally benefitted by a pruning mechanism. Furthermore, a popular specification for Java 1.6 (stemming from the OpenJDK project[Sun09a]) is readily available. Next, comments and whitespaces can be preserved in the lexer's token stream while being excluded from the analysis performed by the parser by redirecting them to a hidden channel.

There exists a specification for Java in versions between 1.4 and 1.6 for the other generators; however, neither JFlex, nor the lexer generators in JavaCC and SableCC provide an easy way to obscure tokens from analysis while still leaving them within the token stream. Also, while the implementation of ASTs can be provided to CUP and JavaCC via additional libraries or own implementation, it is more convenient to use ANTLR's built in systems. Finally, the flexible ASTs generated in ANTLR are easier to handle within the analyzer in comparison to the rigid ASTs generated by SableCC.

Having considered the aforementioned features, both advantageous and otherwise, and as an effect of a comparison to other parser and lexer generators, ANTLR has been selected for the implementation of the Java precompiler tool.

Chapter 5

Implementation

This chapter discusses the implementation of a Java precompilation tool providing the abilities to establish upper bound on the number calls done to specific remote objects by means of static analysis, and to generate source code for the Atomic RMI library using the results of that analysis.

The first section presents the process leading to the development of the analyzer and the project's timeline. The second section details the structure of the system and the employed analysis. The final section shows an overview of the tool's configuration.

5.1 Development

The development of the implementation was planned in an iterative and incremental fashion, where each iteration consisted of a planning phase, an analysis phase, and an implementation phase, and a brief testing phase. Additionally, the entire process was preceded by a prolonged initial planning and an overview of available tools. The development was followed by a final test phase.

The increments in the development were planned and executed as follows.

5.1.1 Initial planning

The initial planning phase was realized from October until December 2008, and consisted of conducting the following activities:

- An analysis of the available theory on static analysis and compiler design for the purpose of defining an initial algorithm for method call analysis;
- An analysis of the documentation to various parser and lexer generators in order to select appropriate tools for the implementation of the Java precompilation tool;
- Implementation of small test projects using some of the parser and lexer generators in order to verify their fitness for the implementation of the Java precompilation tool.

5.1.2 Architectural design

Having specified the technology necessary for the implementation, the following iteration aimed to provide a template for the architecture of the precompilation tool. It was planned out for January 2009, and consisted of the following activities:

- A modular architecture was conceived and planned;
- The foundations of code structure were implemented;

- Source file reading and writing mechanisms were implemented and tested using a generic grammar.

5.1.3 Grammar specification and internal representation design

The iteration was conducted in March 2009 and its goal was to provide a definition and implementation of a language grammar, AST, and analyzer that could be used to create the first working (feature-poor) version of the precompilation tool, but could be easily extended in the future, as new features were added.

- The Java grammatical specification was pruned and redesigned to produce ASTs, relevant to call counting.
- The AST post-processor was designed and implemented.
- The concepts of internal representation and labeling were planned and implemented.
- A basic implementation of the analysis proper was implemented.

5.1.4 Code generation

The iteration was conducted in April 2009 and it aimed to provide a first version of the code generation module which could later be extended adding new features.

- Unique identification mechanisms for objects were conceived and implemented.
- Implementation of fully qualified names for objects.
- A basic version of the code generator was implemented that could add code to the original source files.

5.1.5 Ambiguous and unknown result handling

The iteration was conducted in May 2009 and its purpose was to introduce a system for handling situations when the results of an operation are unknown, when it cannot be established which particular block of code will be executed in case of a choice, or how many times a given block of code will be executed.

- Concept of types and an arithmetic supporting unknown values.
- Implementation of unknown value handling into the existing framework.
- Modification of the analyses to support unknown values and their arithmetic.
- Support for conditional blocks and loops.

5.1.6 Method call analysis algorithm

The iteration was conducted in June 2009 and its goal was to define and document the algorithm for the analysis of the upper bound on method calls to specific objects.

- Concept, discussion, and documentation of the algorithm.
- Introduction of the new algorithm into the existing framework.

5.1.7 Advanced feature support

The iteration was conducted in July and August 2009. It aimed at providing various additions to the currently implemented features.

- Concept and implementation of expression evaluation.
- Algorithm design, documentation, and implementation of loop evaluation count analysis.
- Configuration and extensions of the code generation module.
- Escape analysis for ambiguous code blocks.
- Additional configuration options, multiple file support, configuration of particular blocks for parsing.

5.1.8 Final testing

The iteration was conducted in August 2009 and it aimed at defining final tests and conducting testing of the precompilation tool.

- Definition of test suites and test cases.
- Implementation of automatic tests.
- Testing and analysis of the results.

5.2 Architecture and analyses

The system is composed of a series of modules, each of which is responsible for a part of the analysis process, with specified input and output. Additionally, an operational memory subsystem is passed among a subset of the processes, representing the available deduced context-dependent knowledge about the state of entities appearing within the analyzed AST.

Figure 5.1 depicts the modules of the Code Analyzer and their interaction among one another, and the following sections provide a detailed description of their operation, i.e., the first section describes the scanner and parser modules used for the construction of an Abstract Syntax Tree from source code; the second section presents the post-processor module which simplifies the Abstract Syntax Tree; the third section comprises of a description of conversion of the Abstract Syntax Tree into a set of data objects; the fourth section describes the syntactic analysis main; the fifth following that presents the mechanisms used for final source code generation from the results of the analysis; and the final section presents the mechanisms used to remember the state of various analyses during run-time.

5.2.1 Lexical analyzer and syntactic analyzer

The lexical analyzer and syntactic analyzer are two modules responsible for converting the source code of the analyzed program into an Abstract Syntax Tree. In accordance to a typical architecture of such a processor, there are separate entities working on the lexical and syntactic level: the lexical analyzer, or lexer, transforms the raw text format of the source code into a stream of tokens; the syntactical analyzer, or parser, recognizes patterns in the aforementioned tokens and creates an Abstract Syntax Tree.

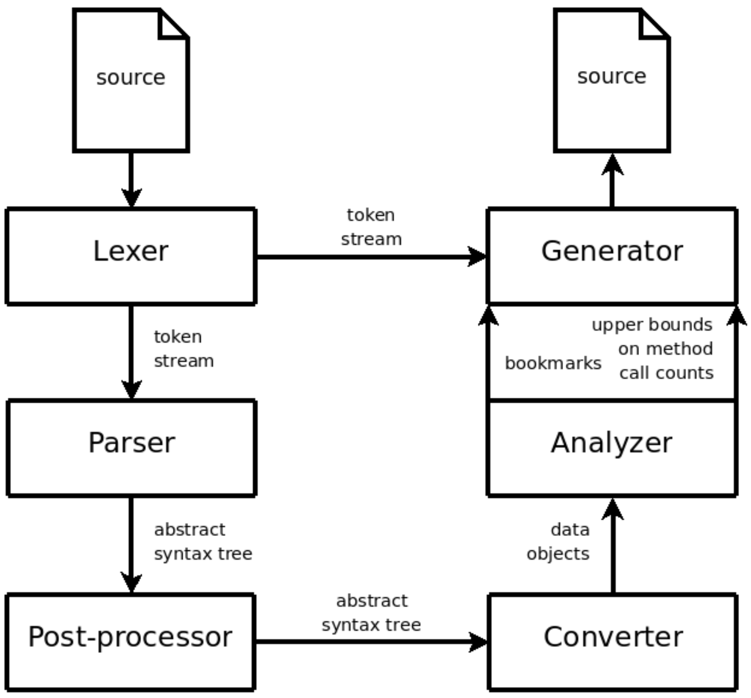

FIGURE 5.1: Modules of the Java precompilation tool.

The lexer and parser are both generated from a single grammar specification using ANTLR. The grammar specification for the components is created basing heavily on the Java 1.6 grammar[Jia09]. The rewrite mechanism was used to morph the parse tree (generated during the analysis of the code by default) into an AST matching the needs of the system. Furthermore, the specification has received alterations, for the parse tree to provide sufficient information about the elements of the code, i.e.:

- declarations of identifiers with a following pair of square brackets throughout the specification were re-written to allow the information about the number of dimensions to be gathered;
- expressions with the same priority, but with two or more operators were redefined to provide the information about which of the operators were used — these expressions consist of: `equalityExpression`, `relationalExpression`, `shiftExpression`, `additiveExpression`, and `multiplicativeExpression`;
- expressions passed to methods are no longer defined as expressions, but instead as arguments — a separate type of entity;
- identifier declarations were further subdivided, so that an identifier is composed of a dot-separated series of identifiers;

5.2.2 Post-processor

The post-processor module is responsible for modifying the AST received from the syntactic analyzer into a form more appropriate in the later stages of the analysis. In particular, the post-processor should introduce a unified form of the AST where ambiguities may eventuate. Furthermore, the post-processor is responsible for providing a simplified version of specific subtrees by replacing certain characteristic branches with placeholders, then moving those subtrees to a different location in the AST, allowing the analysis to be conducted in a more uniform manner.

The architecture of the post-processor is modular, allowing new units to be included to perform additional modifications to the AST, as new needs arise overtime. The modules implemented in the post-processor up-to-date are described in the following sections.

Method declaration rewrite

Method declaration rewrite is a post-processor module whose responsibility it is to provide uniform AST subtrees for declarations of methods that return arrays.

The proposition of reworking such method declarations stems from the fact that two options for declaring methods returning arrays are supported by the Java Language[GJSB05], syntax-wise:

- An array return type can be declared by appending pairs of square brackets to the type name, in front of the method identifier. The number of pairs of brackets is equal to the number of dimensions of the returned array. An example of this type of declaration for a method returning a two-dimensional array is,

```
int[][] arrayMethod() {
    ...
}
```

- An array return type can be declared by prepending pairs of square brackets before the method body is specified. That is, pairs of square brackets may be inserted between the body of the method and its parameters. The number of pairs of brackets is equal to the number of dimensions of the array. This syntax option is obsolete, and only preserved for compatibility with early versions of the Java language. For instance, a method returning two-dimensional array can be declared as follows,

```
int arrayMethod()[][] {
    ...
}
```

The rework has been included during post-processing, since the construction of the parser does not allow for the uniformity to be achieved on the level of syntax analysis without rewriting a significant portion of the base grammar.

The procedure consists of removing the obsolete subtree from each method declaration containing one, and inserting it into the type declaration of the array. In the eventuality where both the method type and the method parameter list are followed by pairs of square brackets, the latter is removed, and the number of dimensions represented in the former is increased by the number of dimensions of the latter. The procedure is illustrated in figure 5.2.

```
1: procedure MethodDeclarationRewrite(abstractSyntaxTree)
2:     for all methodDeclarationTree in abstractSyntaxTree do
3:         removeSubtree(methodDeclarationTree, Array)
4:         if Type contains Array then
5:             Type.dimension ← Array.dimension + Type.dimension
6:         else
7:             insertSubtree(Type, Array)
8:         end if
9:     end for
10: end procedure
```

Figure 5.2: Method declaration rewrite procedure.

Variable declaration rewrite

Variable declaration rewrite is a post-processor module responsible for moving assignments nested withing variable declaration subtrees to a parent node. Such a modification will allow further analyses to be simpler, ensuring that initializations and assignments are always separate statements.

The procedure entails removing assignment subtrees from variable declaration subtrees and inserting them as siblings of that variable declaration subtree, so that an assignment can be processed immediately after a declaration. The procedure is depicted in figure 5.3.

```
1: procedure VariableDeclarationRewrite($abstractSyntaxTree$)
2:     for all $variableDeclarationTree$ in $abstractSyntaxTree$ do
3:         removeSubtree($variableDeclarationTree$, Assignment)
4:         insertSubtree($variableDeclarationTree.parent$, Assignment)
5:     end for
6: end procedure
```

Figure 5.3: Variable declaration rewrite procedure.

Nested call rewrite

Nested call rewrite is a post-processor module responsible for moving method calls nested inside other method calls (as parameters) to a suitable ancestor node of the former. Such a reassignment simplifies the analysis of method calls, which do not need to be recurrent, but instead can deal with simple values.

Additionally, each call in the AST (either nested or otherwise) is given a unique label at this point, representing the result that the call produces. The unique labels are based on a global sequence dispensing unique numbers, concatenated to the end of a string representing the type of the label (i.e., a nested call result). The labels are also inserted into former parents of nested calls to indicate that the result is used in a given place.

The procedure consists of the following steps: first, a unique label is generated for each call subtree in the AST. Then, the closest ancestor node of a suitable type is found — suitable nodes being those representing code blocks or method bodies. If the such closest suitable ancestor is already the parent of the call, than the call is not nested and no further operation is required; otherwise, the call is removed from its immediate parent and inserted into its closest suitable ancestor. The procedure is presented in figure 5.4.

The order of processing calls needs to ensure that the innermost nested calls will be evaluated first in the following analyses, and among themselves, in order of the parameters through which they are passed. Therefore, rewriting the calls must be started from the outermost calls (those which are not nested) to the innermost ones. To ensure that the calls are evaluated in the order of passing to a method it is required to rewrite them from the leftmost sibling to the rightmost one (lowest to highest index).

Initialization rewrite

Nested call rewrite is a post-processor module responsible for moving method calls nested inside initialization subtrees. The principle of operation is exactly the same as that of the nested call rewrite module, described in the previous section.

```
procedure NestedCallRewrite($abstractSyntaxTree$)
    for all $callTree$ in $abstractSyntaxTree$ do
        $resultTree \leftarrow$ getNextUniqueLabel
        $closestSuitableAncestor \leftarrow$ findClosestSuitableAncestor($callTree$)
        if $closestSuitableAncestor \neq callTree.parent$ then
            removeSubtree($callTree.parent$, $callTree$)
            insertSubtree($callTree.parent$, $resultTree$)
            insertSubtree($closestSuitableAncestor$, $callTree$)
        end if
    end for
end procedure
procedure findClosestSuitableAncestor($callTree$)
    $node \leftarrow callTree$
    loop
        $node \leftarrow node.parent$
        if $node$ is in {Block, Body, Nil} then
            return $node$
        end if
    end loop
end procedure
```

Figure 5.4: Nested call rewrite procedure.

5.2.3 Converter

The converter is responsible for interpreting the various subtrees found in the AST into a set of interconnected data objects. The goal of this transformation is to provide an interface for the main analysis to access data, without the need to perform operations on trees. Additionally, the converter is able to cast the various data in the tree into their proper type.

The converter is composed of modules, or *extractors*, which recursively retrieve specific data objects from subtrees that adhere to specific patterns that signify various types of code blocks. These extractors are structured into a hierarchy corresponding the class hierarchy of the data objects they produce, and therefore, roughly similar to the structure of the AST.

5.2.4 Semantic analyzer

The semantic analyzer is the main module of the system. Its purpose is to discern a set of remote objects that are used in the code, and to establish an upper bound on the number of calls to each remote object, by conducting analyses on the provided data.

The semantic analyzer is also required to recognize the elements of the Atomic RMI API. This information collected by the semantic analyzer is useful to the code generator module, which will be able to locate appropriate places in the code where the information about the number of calls to remote objects needs to be included.

The analysis conducted by the module is a combination of variants of a points-to analysis, an escape analysis, and a side-effect analysis:

- The elements of points-to analysis keep track of what set of memory locations are likely to be pointed to by a given variable at any given point in time;
- The elements of escape analysis try to establish whether the values of any given variable can be predicted;
- The elements of side-effect analysis predict when expressions modify objects outside of their scope.

Additionally, loop expressions are analyzed to predict if they are going to evaluate. In the future, analysis with the use of symbolic analysis, in an attempt to find out how many times the loop will evaluate exactly can be implemented instead of this simplified analysis.

The semantic analysis is modular and it is applied to various objects individually, with only a context object and a result object passed between levels. The individual components responsible for analysis of specific data objects (derived directly from subtrees of the AST) form a structured hierarchy.

The details of counting calls can be found in the description of the MCCA in section 3.3. The following sections roughly explain the individual analyses, concentrating on analyses other than call counting.

Assignment analysis

Assignments are expressions, where a value is assigned to a variable. The analysis of such expressions means to track what values the variables can potentially be assigned. The results of that analysis are passed to the context.

If an assignment was declared in the code, a potential write to objects from outside the current scope can be performed — assignment analysis also attempts to discover such situations and notify the results of their occurrences. Writes can also cause objects to escape, if a value which has also escaped is assigned, or if it cannot be predicted whether the assignment will happen during evaluation, or precisely what value will be assigned.

Declaration analysis

Declarations are expressions announcing a variable will be used in further source code as an instance of a particular type, and with a given identifier. The analysis of declarations aims to provide the information about variables to further analyses, such as type and modifiers (e.g., whether a variable is final). The results of the analysis are passed to the context.

Call analysis

Method calls are expressions used to send a message to an object or class, that request a certain method to be executed. The analysis of method calls mainly concentrates on updating the state of the various method call counters.

Additionally, method call analysis consists of several modules that are applied to specific function calls related to the Atomic RMI API, and used to distinguish certain fragments of the code; specifically, to find out where each of the Atomic RMI tasks begins and ends. The results of these analyses are often bookmarks – references to a particular range of tokens in the token stream, with a snapshot of the state of the analysis at that point. Furthermore, the modules are used to create unique identifiers for remote objects.

Finally, method call analysis tries to establish whether any parameters passed to the call are escaped. This is assumed when the body of the method is not available for analysis and the method is not listed as a pure method (without side-effects), or if the parameters are modified, but the new value is not known.

Method body and block analyses

For the purpose of the analysis, blocks of code are series of expressions sharing a common context.

Block and method body analyses are conducted by recursively conducting an analysis of the expressions contained within them.

Within the analysis, each nested block operates on its own context, which is linked to the context of its parent block. This prevents information unknown to the parent block from being overwritten.

Apart from ordinary blocks of code, such as method bodies and explicitly declared scopes, the following classes of additional analyses can be identified, for types of blocks with certain characteristics:

Conditional block analysis

Conditional blocks are blocks which may be evaluated during program run-time if a guard condition evaluates to a boolean value of true. Optionally, an alternative block may be present, which is evaluated when the guard condition evaluates to *false*.

The analysis of such blocks required an attempt at evaluation of the guard condition. Depending on what values were used to express the conditions, the outcome of a condition may be established as either *true*, *false*, or *uncertain*. In the foremost case the guarded block is analyzed. In the case of an evaluation to false the guarded block is ignored, and if an optional alternative block is defined, then it is analyzed.

In the case of an uncertain evaluation, where the result will depend on the status of variables at a given point during the execution of the program the following procedure applies:

- As regards call counts to specific objects, the highest possible number of calls should be counted for the entire structure, meaning that the guarded block should always be analyzed and taken into account; if an alternative block is present, it should be analyzed, and, the highest count from the two blocks should be taken into account.
- As regards tracking identities of objects, any modification to variables is uncertain, and should be marked as such. This means that any objects which are potentially altered by any of the blocks should be discarded as escaped pointers, unless it can be established that the objects will retain the same value if either of the blocks were executed.

Loop block analysis

Loop blocks are blocks evaluated repeatedly during the execution of the program. The number of execution times is typically based on a guard condition, unless an instruction forces early termination.

The number of executions can range from none to an infinite number. Therefore, the analysis aims at predicting the maximum number of times the looped block may be executed during run-time, using the guard condition as a basis for this estimation in conjunction with the current predicted context. If no estimate can be given it is necessary to assume that an infinite number of executions can be the upper bound. Break statements can be ignored at this juncture, as they are unable to cause additional executions, only to limit their number.

Additionally, write operations performed from within the loop cause uncertainties to arise, unless it can be predicted that a constant value is written to a particular variable, and, therefore, it does not depend on the number of iterations. As such, any objects which are potentially altered by the block in an unpredictable way should be discarded as escaped pointers.

5.2.5 Code generator

The code generator is a module responsible for producing source code on the basis of the results of the analyses. The module uses the token stream from the lexical analyzer as input.

The generation of the code takes place when a bookmarked token is found in the source token stream. The bookmark provides the information on what variable names should be used in the output to reference specific memory locations.

To maintain aestheticism, the indentation of the original source code is established by the generator and preserved in the generated source code fragments. To discern the indentation necessary for new code, the existing code is analyzed, meaning that if the present style of programming does not adhere to a strict standard regarding indentation (e.g., that specified in [Sun99]) the generated code may not be aesthetically pleasing.

The generator does not override relevant code annotated as inserted manually, allowing the developer to offer a better estimation on the upper-bound number of calls to remote objects.

Apart from generating code, the generator recreates the source program from the input stream without alteration.

5.2.6 Memory

The various modules operate largely independently of each other (the modules depend on one another only indirectly, through the data each of them expects as input); however, there is a need to share certain data structures among them. The storage of information passed between the modules and individual analyses is done twofold: through a context and via results.

Context

The context is a data structure serving as the immediate memory accessible by various analyses within the main semantic analysis module. The main purpose of the context is to store information about the state of variables, recognized memory locations, and predicted states of entities appearing in the source code at any given time. A context pertains to a block of source code and the blocks nested within it.

The context is used to store the information about:

- currently declared variables and their values,
- known identifiers and their types,
- imports and fully qualified names of known types,
- information whether known types are passed by value or reference,
- results of calls.

The context is hierarchical in nature: when an analysis descends to a new block, the context is forked, and a new copy is passed to the analysis of the inner block. This causes the lower analyses to be able to register their own variables and data without overriding the data from the parent block or leaving a trace of methods that should be inaccessible in the parent block. The data carried from the parent can still be passed to the child, allowing modification.

Certain elements of the context can be loaded before the analyses are performed, from a pre-prepared text file. This is meant to allow a way to provide accurate information about certain inaccessible parts of the system and, especially, the Java APIs. Loading this information beforehand ensures that the analyses have a basic toolkit to recognize standard types and their basic characteristics.

Results

Results are a data structure containing the information on the information deduced from the analysis; therefore, this structure collects the information which are globally valid for the entire process.

Results consist of the following information:

- the number of calls to each memory location (object),
- the number of calls to each variable identifier,
- information on escaped objects and performed writes,
- information about the currently analyzed task,
- bookmarks.

Results are a monolithic structure, passed to analyses with the permission to apply changes directly.

A bookmark is a structure devised to reference specific method calls in the code, with a number of information useful in later generation of the output in the generator module.

5.3 Configuration

The Java precompilation tool uses three types configuration: property files, directories containing files that function as lists, and code annotation. The tool is also able to function in a situation where no configuration files are present (in the case of property files, this means that default values are used by the system). All of the aforementioned means are used to parametrize the tool at runtime.

The following configuration areas are available to the developer working with the system.

5.3.1 Classes and type information

If the code uses objects and methods which are predefined, or well known to the developer, certain attributes can be assumed. These include the knowledge whether the object is identified by value or by reference, and what is the fully-qualified name of a type.

The user can configure a number of details relating to such cases. That is, lists of fully-qualified types can be manually provided using the configuration. Additionally, lists of objects passed by value can be defined by the developer in advance.

5.3.2 Code annotation

In an event when program source is analyzed manually and an upper bound on the number of method calls is declared, an annotation should be made in the source code as a comment following that line, and containing a predefined keyword. By default the keyword is set to `$MANUAL$` and can be modified in the general configuration. Using this annotation prevents the code generator from overriding that line of code.

5.3.3 General configuration

The precompilation tool can be provided with general configuration, defining paths pointing to other configuration resources (e.g., the directories containing list of fully-qualified types), annotation keywords, and the output preferences for the tool.

Additionally, configuration is available for the logging mechanism used in the system (based on Log4j[Apa07]) with the ability to activate or deactivate it, or configure the precise information and mode of logging.

Chapter 6

Testing

The software tests presented in this chapter are meant to verify the correct operation of the static analyzer in a number of scenarios concentrating on the correct analysis of method calls and proper generation of source code. The test cases are divided into several separate test suites, each pertaining to a particular feature, and each composed of multiple test cases.

The first section contains the descriptions of test suites and their constituent test cases. The second section details the technology used for testing. The final section presents the results of the conducted tests.

6.1 Test details

This section details the test suites containing an introduction explaining the main goals of the suite, as well as the general methods for obtaining those goals, followed by a list of test cases of which it is formed. Each test case is composed of a unique identifier, a descriptive name, and a statement of purpose.

6.1.1 Simple expression analysis test suite

The purpose of the simple expression analysis test suite is to verify the correct operation of the simplest analyses that contain a straightforward sequence of calls, a single call, or no method calls to remote objects. These tests are meant to provide confidence that the most basic elements work correctly for the simplest of cases.

SET1: Analyze a single method call

The test case verifies whether a single remote object method call is registered satisfactorily by the analyzer, and that the proper code is created by the generator.

SET2: Analyze a sequence of method calls on multiple variables

The test case ensures that a series of sequential calls is analyzed properly, and that the adequate code is generated.

SET3: Analyze a sequence of method calls on variables changing objects

The test case checks if switching the remote objects that variables reference have any adverse effects on the operation of the analyzer; additionally, it is tested that the correct code is generated in such an eventuality,

SET4: Analyze code with no method calls to remote objects

The test case confirms, that in the event of no method calls to remote objects, both the analyzer and the generator act appropriately.

6.1.2 Block expression analysis test suite

The purpose of this test suite is to verify that blocks of code nested within other blocks are analyzed correctly. These tests aim to provide a certainty that, when the recursive analyzer traverses the tree of nested blocks, the estimated number of method calls will be passed along as the analysis of child nodes is conducted, and that the values of variables that undergo changes within the nested blocks are applied as well to the parent nodes and siblings. To serve the former purpose, method calls are issued to objects within the nested blocks, and for the latter, a number of assignments and declarations are entered into the tests, that would interfere with method counting, if not interpreted correctly.

BET1: Analyze method calls in a nested block

The test case verifies whether the analysis of method calls in a single nested block is conducted adequately, and whether the resultant source code is generated properly.

BET2: Analyze method calls in multiple nested blocks

The test examines the analysis of method calls issued to remote objects in a multiply nested blocks, and verifies that the generated code is correct.

BET3: Analyze method calls in a nested block on variables changing objects

The test case establishes that an analysis of multiply nested blocks yields correct results in an event when the variables referencing remote objects are pointed to different objects at different levels of the AST; in addition, the generation of source code in such cases is verified.

6.1.3 Conditional block expression analysis test suite

The goal of this test suite is to verify the correct analysis of conditional expressions, whose condition can be evaluated to a specific known value. The tests' purpose is to prove that only the adequate fragment of code will be analyzed for method call counts. To this end, four tests are conducted that check the correctness of analysis for conditional expressions with conditions evaluating to both *true* or *false*, and both conditional expressions that do or do not posses alternative blocks. Additionally, the mechanisms for establishing the values of conditions are tested.

CET1: Analyze method calls in a conditional block if the condition evaluates to true

The scenario tests whether a simple conditional block evaluating to *true* is analyzed correctly, and the appropriate code is created for the results of its analysis.

CET2: Analyze method calls in a conditional block if the condition evaluates to false

The test case checks whether a simple conditional block evaluating to false is analyzed correctly (i.e. ignored) and if the correct output code is generated.

CET3: Analyze method calls in a conditional block with an alternative block if the condition evaluates to *true*

The test case verifies if a conditional block with an alternative block (an *else* clause) is capable of being analyzed in terms of the number of method calls to remote objects, and whether the generate source code meets expectations.

CET4: Analyze method calls in a conditional block with an alternative block if the condition evaluates to false

The test ensures that an alternative block is analyzed correctly, if a condition is evaluated to false, and whether the correct source code is generated on output.

6.1.4 Loop block expression analysis test suite

The aim of this test suite is to ensure the the loop blocks with an evaluable number of iterations are analyzed correctly. The suite's constituent test cases attempt to establish that the analysis recognizes whether the loop guard condition disallows execution of the loop, or, if the condition may be evaluated to *true*, that a safe number of iterations is registered. Additionally, the analysis of nested loops is verified.

In the case of the simplified algorithm for the estimation of loop iterations, it is assumed that if the loop's condition evaluates to *true*, the loop may be evaluated a maximum number of times equal to infinity, and this is represented in the tests.

LET1: Analyze method calls in an unevaluated loop block

The test case checks whether the analysis and code generation is correct for a loop which can never be entered.

LET2: Analyze method calls in a loop block evaluated once

The test case analyses the correctness of the system in a situation where a loop may be entered and evaluated once. The analyzer is expected to arrive at the answer, that the loop may be evaluated an unknown number of times.

LET3: Analyze method calls in a loop block evaluated n-times

The test case establishes whether a loop that is evaluated a certain number of times is analyzed correctly, and whether the appropriate source code is generated for that situation. The analyzer is expected to suggest that the loop may be evaluated an unknown number of times.

LET4: Analyze method calls in a loop block evaluated an infinite number of times

The test case verifies if a loop with a condition that always evaluates to *true* will be analyzed an an appropriate fashion, and whether the appropriate source code is created on output.

6.1.5 Ambiguous evaluation test suite

The purpose of the ambiguous evaluation test suite is to verify the behavior of the analyzer in an event when the value of a variable, or an expression is unknown or uncertain, leading the analysis to be unable to count method calls exactly, but instead, needing to find an upper bound on the possible number of calls. The tests proceed by testing the structures whose evaluation depends on conditions: loops and conditional blocks.

AET1: Analyze method calls in a conditional block, where the condition is unpredictable

The test case means to ensure that the conditional block is analyzed correctly (the block is assumed to be potentially evaluated) in an event where the value of the condition is unknown; it is also checked if the generated source code is appropriate.

AET2: Analyze method calls in a conditional block, with an alternative block, where the condition is unpredictable

The test case tries to verify whether the analyze produces proper results when encountering a conditional block whose condition is unknown, and which has an alternative block. The test case also checks if the created source code is complete and correct.

AET3: Analyze method calls in a loop block evaluated an unpredictable number of times

The test case verifies whether the analysis of loop blocks with uncertain conditions, and therefore an unknown potential number of evaluations, produces the correct results and output source code.

6.1.6 Code generation test suite

The purpose of this test suite is to verify functions of the source code generation module: the ability to successfully write over previously generated results without confusion, and the ability to keep manually inserted values of method call counts. The tests produce conditions in which the generator's functions may be tested, and verifies the created source code.

CGT1: Generate source code with manually preset results

The test case aims to check if the source code inserted manually by a developer (with the appropriate annotation) remains after an analysis and generation of source code.

CGT2: Generate source code with over prior generated results

The test case verifies that the source code generated in a previous analysis is not an obstacle to a successful analysis and generation in a subsequent analysis of the output file.

6.1.7 Multiple entity evaluation test suite

The multiple entity evaluation test suite is designed to test the ability of the system to analyze and generate multiple files, which may consist of several analyzed method, which in turn may contain a multitude of defined tasks. The test suite analyzes each of these situations and verifies whether the resulting code is generated appropriately.

MET1: Analyze method calls for a method with multiple tasks

The test case analyzes the processing of a single file containing a single method, in which a number of tasks may be found – some nested, or evaluated conditionally. The test case verifies that the appropriate source code is generated for each of the tasks.

MET2: Analyze method calls for a class with multiple methods

The test case confirms that the analysis of two methods in a single file yields appropriate results and produces correct source code. Additionally, the methods both contain multiple tasks.

MET3: Analyze method calls for a set of classes

The test case is responsible for the verification of a number of analyses performed on a set of files, each of which may contain multiple methods composed of complex structures and multiple files.

6.2 Testing technology

The system-wide tests were automated using the JUnit 4.7 library[JUn09] – a popular testing framework for the Java language, designed to support various classes of tests.

In general, a data-driven approach to testing was pursued, where each test case is conducted by obtaining a program source code containing the feature under test, running the analysis, and comparing the source code of the generated file to a file containing the expected source code – if the files are not different, with the exception of whitespace, the test is considered to be passed.

6.3 Results

The final tests were conducted on 30th August 2009. A total of 22 test cases were run, out of which 19 were completed successfully, and 3 failed. The results are presented in table 6.1.

Test suite	Test case	Result
Simple expression analysis	SET1	Passed
	SET2	Passed
	SET3	Passed
Block expression analysis	BET1	Passed
	BET2	Passed
	BET3	Passed
Conditional block expression analysis	CET1	Passed
	CET2	Passed
	CET3	Passed
	CET3	Passed
Loop block expression analysis	LET1	Passed
	LET2	Passed
	LET3	Passed
	LET4	Passed
Ambiguous evaluation	AET1	Passed
	AET2	Failed
	AET3	Failed
Code generation test suite	CGT1	Passed
	CGT2	Passed
Multiple entity evaluation	MET1	Passed
	MET2	Passed
	MET3	Failed

TABLE 6.1: Test results.

The unsuccessful finalization of test cases AET2 and AET3 is a result of an incorrect implementation of the join operator in the analysis of conditional block. The failure of test case MET3 is a result of improper generation of the source code (form correct data), in an event when multiple tasks are present and a manual value has been defined for method calls in one of them. However, both of these problems stem from faulty implementation, and not incorrect concept. Plans for quick removal of the problems have been conceived.

The summary report of the code coverage (the portion of the code that is run during execution) of the tests is presented in table 6.2. It can be said that the coverage is generally satisfactory. The code not covered by the tests is typically used in debugging, prepared for additional future functionality in the form of a rough framework, or a part of general-purpose toolkits.

Class coverage	Method coverage	Block coverage
94% (165/175)	76% (804/1060)	74% (11424/15439)

Table 6.2: Test code coverage.

Chapter 7

Related work

The author is unaware of other works that aim to conduct static analysis of program sources on the semantic level and insert additional lines of code into those programs without the necessity of prior annotation. Especially, the author is unable to point to projects aiming to generate code on the basis of method call count information obtained via static analysis. There are however certain types of applications which perform a similar sort of task, or whose general manner of working bears some resemblance to those of certain individual parts of the Java precompilation tool; specifically, its analyzer module and its generator. Such tools, or rather classes of programs, are outlined in this chapter: the first section presents automated library binding generation software, the second section provides an overview of software used for source code verification through static analysis, and the final, an example of a Java precompiler used for generating assertions.

7.1 Library binding generators

The software that performs a similar type of task to the Java precompilation tool are programs which assist programming language interoperability by automatically generating bindings to libraries. Such bindings allow the developer writing an application in one programming language to use the libraries from other languages without having to reimplement their functionality.

The end goal of the process is to create a library in the target language that provides a wrapper for the source library: the resultant library mimics the original function signatures (with allowances for the peculiarities of both the source and target languages), and its purpose is to invoke the functions of the source library, providing the necessary type conversions and additional actions both prior and after the execution.

In a simpler form of library binding generation, such as that employed by SWIG[Bea96] and GlueGen[Sun09b], only function definitions (e.g. headers) and rules regarding type conversions are necessary on input. However, if the output library is meant to take advantage of concepts particular to the target language, but not present in the source language, the original source code needs to be analyzed in its entirety. Such analysis is meant to find function signatures and the various side effects that the invocation of functions can cause (e.g. a global object is allocated or an error number is set in a global variable), as well as provide information for automatic structure conversion.

The latter form of gathering information is kin to the analyses performed by the Java precompilation tool, and an example of such a procedure can be seen in the Python binding generator for the C language, presented in [RJAL09]. This program conducts a number of inter-procedural, context-insensitive analyses on the code of the source library in order generate functions in an idiomatic Python form (e.g. using tuples instead of output parameters in case of functions returning multiple values), and to create wrapper objects that could benefit from Python's garbage collection mechanism. these analyses include the following:

- Output parameter analysis — a forward dataflow analysis whose purpose is to find which of the function's parameters can be classified as output parameters (pointer-type parameters which are used to pass values to the outside of the function, which characteristically are written to before they are read).
- Array analysis — a method of discerning which pointers are used as arrays. The analysis is performed in two steps: the analysis of interprocedural information propagation and the search of local use of the pointer in an array context.
- Resource management analyses — a means of gathering information about how various resources should be used in a given library in terms of the memory management system of the target language. These analyses include an analysis of pointer ownership, and an escape analysis. Furthermore, constructors and finalizers are detected by forward dataflow schemes.

7.2 Static code verification

The Java precompilation tool shares certain aspects with programs whose purpose is to conduct static analyses of source code on the semantic level in order to ensure that the processed sources are free of specific classes of erroneous behavior — a goal achieved through the application of dataflow schemes and partial evaluation. Certifying a program in such a way provides an increased level of trust in the success of its continued operation – an especially important concern in critical systems.

A number of applications, like Findbugs[APM$^+$07], use a heuristic approach, where a number of bug patterns are attempted to be uncovered in the program under scrutiny; however, there exist systems where an exact analysis is applied instead.

An example of a source code analyzer with uses in code verification is the value analysis plug-in for Frama-C[CP09], a framework for collaborative exact (non-heuristic) static analysis and verification of C programs. The value analysis plug-in may be directly employed to infer the absence of run-time errors or predict a set of values a variable may hold at any point in the program. The analyzer's mode of operation features several noteworthy features, including:

- Loop analysis allowing for approximations in an event when the evaluation of loops poses a threat of not completing (e.g., due to unknown values affecting the condition). Such a mechanism allows the analyzer to process loop blocks in finite time.
- Analysis of functions by direct expansion at the point of invocation, providing a precise result, although preventing the method from providing support for recursive functions.
- Partial application support — analyses of programs whose source code is not complete, or which use library functions with no implementation available for static analysis. The solution to the problems posed by the inaccessible sources are: making concessions in the assumptions of global values at entry point, and allowing the developer to make annotations regarding fragments of code which have well known behavior.

7.3 Assertion generation

Another major aspect of the Java precompilation tool is, as with all precompilers, the generation of code, or rather, the ability to inject short, compilation-ready code fragments into existing programs. Precompilers have an abundance of uses, however the most prevalent is their utilization in generating code for the enforcement of assertions – a mechanism stemming from the approach to software design known as *Design by Contract*.

Jass[BFMW01] (Java with Assertions) is a tool allowing to define contracts in the form of method invariants, pre-conditions, post-conditions, and other assertions in Java programs. This functions are realized by a precompilation tool which is configured by annotations, where the specification of contracts is defined in the Java Modeling Language[CKLP06] (JML), and contains the declaration of the assertion type and the conditions which must be met. The conditions are expressed using Java language (they are copied into the output program), and they may be checked for side effects with a dataflow analysis, if they contain method invocations. Then, the precompiler creates and injects source code that checks the specified conditions during runtime and prevents the method from executing if these requirements are not met.

Chapter 8

Conclusion

8.1 Accomplished goals

The proceedings of the development and the implemented tool (as presented in chapters 3 and 5) have been successful at limiting the amount of effort needed for the programmer to develop Atomic RMI programs. This has been achieved by providing a means to automatically generate atomic task headers, which contain the specification for a maximum expected number of method calls within that task to individual remote objects. Indeed, the programmer is relieved both from the necessity to analyze source code in an attempt to acquire the information required to create task headers, as well as the need to implement the headers.

Besides, a number of specific features are supported in the implemented precompilation tool, among the most important of which, comes the ability of the tool to analyze and produce correct results for all Java programs that can be represented using SL (presented in section 3.2 and appendix A). Additionally, the analysis can handle extra constructs, i.e., arithmetic expressions, and bodies of called methods.

Furthermore, the tool generates execution-ready code, so that it can fulfill its primary role of a precompiler, i.e., it does not necessitate additional user input after the generation has been completed). In addition, the generated code retains human-readability and attempts to use an identical indentation scheme as the original program. Also, the program is able to analyze the code that it has once generated with correct results.

Moreover, the tool is designed using a modular architecture, meaning that it is possible to replace or modify any of the main components, especially, the lexical and syntactic analyzers, or the code generator, to provide support for, a different SL-compatible programming language, for example. Beyond that, the framework of the semantic analyzer is designed in a way to facilitate adding support to new construct, thus reinforcing the extendability of the precompiler.

It is also important that the precompilation tool allows the developer to override the generator manually by annotating the source code prior to analysis. The annotation allows the process to be fine-tuned in specific cases, where the developer is in possession of facts unknown to the analyzer.

Finally, the precompilation tool provides configurability in respect to logging during the various stages of processing, the nature of predefined classes, and other properties.

8.2 Further work

Though the main goals of the thesis has been fulfilled, one may find a number of areas — such as the supported language constructs, — that can be developed further. Due to the scale of the possible developments, these improvements could not be implemented as a course of this thesis.

First and foremost, although the necessary framework for such an extension is provided, future efforts may be expanded to provide support for the entirety of the Java language, which was not undertaken in the course of this thesis due to the enterprise's time- and effort-consuming nature, or another programming language that is a superset of SL. Among the most valuable elements that may be introduced to the system are arrays, exception blocks, and switches-case conditional blocks.

Secondly, the current framework may be broadened to inter-operate with libraries other than Atomic RMI, providing they required information about maximum method call counts before the execution of the program. Currently, the precompilation tool integrates its generated code solely with Atomic RMI due to a need for conducting an additional analysis and recognizing the characteristic elements of the library's API. Similarly, other libraries would require dedicated analyses.

Furthermore, the current framework may be equipped with extra analyses, working alongside the main semantic analysis. The role of those analyses would be to provide supplementary details, making the estimation of upper bounds on method calls to remote objects more precise, where possible. Examples of such analyses are a more reliable analyzer of loop evaluation counts or a more accurate pointer analyzer.

Finally, the syntactic and lexical analyzer, and the code generator could be exchanged and reconfigured to provide support for other languages, that bear similarity to Java and can be expressed using SL, such as C♯.

To summarize, the future introduction of the presented improvements would undoubtedly result in improving the capabilities of the precompilation tool or providing it with additional features.

Appendix A

Complete EBNF specification for SL

The following EBNF listing provides a complete description of SL,

```
expression = block
           | conditional_block
           | loop_block
           | variable_declaration
           | object_initialization
           | variable_assignment
           | method_call;

variable_declaration = type, variable;
object_initialization = type, { argument };
variable_assignment = variable, expression
                    | variable, literal
                    | variable, variable;
method_call = variable, method;

block = { expression };
conditional_block = condition, positive_block,
                    [ alternative_block ];
loop_block = condition, body;

positive_block = block;
alternative_block = block;
condition = expression | literal;
body = block;

method = identifier, { argument };
argument = identifier | literal;
variable = identifier;
type = identifier;

literal = BOOLEAN;
identifier = ALPHABETIC_CHARACTER { ALPHANUMERIC_CHARACTER };

BOOLEAN = 'true' | 'false';
DIGIT = '0' ... '9';
ALPHABETIC_CHARACTER =  'a' ... 'z' | 'A' ... 'Z' | '_';
ALPHANUMERIC_CHARACTER  = ALPHABETIC_CHARACTER | DIGIT;
```

The all-uppercase productions represent tokens (one or more terminal characters). The symbol ... is a shorthand notation used here to describe a range of characters, e.g. the definition of token *DIGIT* is any character from the range of characters from *'0'* to *'9'* – all the digits.

Bibliography

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools.* Addison Wesley, 2nd edition, August 2006.

[Apa07] Apache Software Foundation. *Logging Services*, September 2007. `http://logging.apache.org/`.

[APM$^+$07] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8, New York, NY, USA, 2007. ACM.

[Bea96] David M. Beazley. SWIG: an easy to use tool for integrating scripting languages with C and C++. In *TCLTK'96: Proceedings of the 4th conference on USENIX Tcl/Tk Workshop, 1996*, pages 15–15, Berkeley, CA, USA, 1996. USENIX Association.

[BFMW01] Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim. Jass – Java with assertions. *Electronic Notes in Theoretical Computer Science*, 55(2), 2001.

[CKLP06] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures, Lecture Notes in Computer Science*, volume 4111, pages 342–363. Springer Verlag, 2006.

[Con79] Control Data Corporation. *CDC Algol-60 Version 5 Reference Manual*, 1979. `http://www.lrz-muenchen.de/~bernhard/Algol-BNF.html`.

[CP09] Pascal Cuoq and Virgile Prevosto. *Frama-C's value analysis plug-in.* CEA LIST, Software Reliability Laboratory, 2009. `http://frama-c.cea.fr/download/frama-c-manual-Lithium-en.pdf`.

[Eck03] Bruce Eckel. *Thinking in Java.* Prentice Hall PTR, 3rd edition, 2003.

[Gag98] Étienne Gagnon. SableCC, an object-oriented compiler framework. Master's thesis, School of Computer Science, McGill University, Montreal, March 1998. `http://sablecc.sourceforge.net/downloads/thesis.pdf`.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java^{TM}– Language Specification, Third Edition.* Addison-Wesley, June 2005.

[HMJH06] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable Memory Transactions. `http://research.microsoft.com/~simonpj/papers/stm/stm.pdf`, August 2006.

[Hud06] Scott E. Hudson. *CUP User's Manual.* Graphics Visualization and Usability Center, Georgia Institute of Technology, March 2006. `http://www2.cs.tum.edu/projects/cup/manual.html`.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[Int96] International Organization for Standardization. *ISO/IEC 14977:1996(E) Information technology – Syntactic metalanguage – Extended BNF*, 1996. `http://standards.iso.org/ittf/PubliclyAvailableStandards/`.

[Jia09] Yang Jiang. ANTLR-based java grammar from OpenJDK project to develop an experimental version of the javac compiler based upon a grammar written in ANTLR. `http://openjdk.java.net/projects/compiler-grammar/antlrworks/Java.g`, January 2009.

[Joh75] S. C. Johnson. YACC – Yet Another Compiler Compiler. Technical Report Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[JUn09] JUnit, August 2009. `http://www.junit.org`.

[Kle09] Gerwin Klein. *JFlex User's Manual*, January 2009. `http://www2.cs.tum.edu/projects/cup/manual.html`.

[Les75] M. E. Lesk. Lex – a lexical analyzer generator. Technical Report Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[Mam08] Mariusz Mamoński. Rozszerzenie mechanizmu zdalnego wywołania metod w językach typu Java o własność atomowego wykonania rozproszonych metod. Master's thesis, Poznań University of Technology, Poznań, Poland, 2008.

[Par97] Terence J. Parr. *Language Translation Using PCCTS and C++*. Automata Publishing Company, 1997.

[Par07] Terrence Parr. *The Definitive ANTLR Reference – Building Domain-Specific Languages*. The Pragmatic Bookshelf, May 2007.

[RJAL09] Tristan Ravitch, Steve Jackson, Eric Aderhold, and Ben Liblit. Automatic generation of library bindings using static analysis. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 352–362, New York, NY, USA, 2009. ACM.

[Som96] Ian Sommerville. *Software Engineering*. Addison Wesley, 5th edition, 1996.

[Sun99] Sun Microsystems. *Code Conventions for the Java Programming Language*, April 1999. `http://java.sun.com/docs/codeconv/`.

[Sun03a] Sun Microsystems. *JavaCC*, January 2003. `https://javacc.dev.java.net/`.

[Sun03b] Sun Microsystems. *JavaCC: JJTree Reference*, January 2003. `https://javacc.dev.java.net/doc/JJTree.html`.

[Sun08a] Sun Microsystems. *Java Remote Method Invocation – Distributed Computing for Java*, August 2008. http://java.sun.com/javase/technologies/core/basic/rmi/whitepaper/.

[Sun08b] Sun Microsystems. *The Java Tutorials. Lesson: Concurrency*, 2008. `http://java.sun.com/docs/books/tutorial/essential/concurrency/index.html`.

[Sun09a] Sun Microsystems. *OpenJDK*, May 2009. `http://openjdk.java.net/`.

[Sun09b] Sun Microsystems Game Technology Group. *GlueGen Manual*, August 2009. `https://gluegen.dev.java.net/source/browse/*checkout*/gluegen/trunk/doc/manual/index.html`.

[UCL05] UCLA Compilers Group. *Java Tree Builder*, 2005. `http://compilers.cs.ucla.edu/jtb/`.

[Woj07] Paweł T. Wojciechowski. *Language Design for Atomicity, Declarative Synchronization, and Dynamic Update in Communicating Systems*. Wydawnictwo Politechniki Poznańskiej, Pl. Marii Skłodowskiej-Curie 2, Poznań 60-965, Poland, 1st edition, 2007. 204pp.

[Woj08] Paweł T. Wojciechowski. Extending atomic tasks to distributed atomic tasks. In *Proceedings of the Workshop on Exploiting Concurrency Efficiently and Correctly* $(EC)^2$ *(co-located with CAV '08: the 20th International Conference on Computer Aided Verification, Princeton, USA)*, July 2008.

[WRS04] Paweł T. Wojciechowski, Olivier Rütti, and André Schiper. SAMOA: A framework for a synchronisation-augmented microprotocol approach. In *Proceedings of IPDPS '04: the 18th IEEE International Parallel and Distributed Processing Symposium (Santa Fe, USA)*, April 2004.

Poznań University of Technology
Faculty of Computer Science and Management
Institute of Computer Science

Typeset using LaTeX in Computer Modern.

BibTeX:

```
@mastersthesis{ key,
    author = "Konrad Siek, 71747",
    title = "{Design and implementation of a Java source code precompilation tool for static
analysis and modification of programs for the Atomic RMI library}",
    school = "Pozna{\'n} University of Technology",
    address = "Pozna{\'n}, Poland",
    year = "2009",
}
```