ABSTRACT
A proof-of-concept Java/MPJ to Promela translator is presented as a step towards verification of safety properties of parallel Java programs using the MPJ library. The translator leverages the Soot library to achieve a three-step method for of the Java to Promela.

Keywords
Parallel computing, MPJ, Control flow abstraction, MPI, model checking

1. INTRODUCTION
Verification of distributed programs continues to be an important field. The industry standard for communication in distributed programs is the MPI standard. Although much work has been published concerning the verification of MPI-based programs, little work has been done towards verification of its Java counterpart: MPJ. This paper presents the design of a tool that can translate Java using MPJ to a corresponding model checker model. The main focus is to abstract the control flow of MPJ logic from the Java program, and verifying that with readily available model checkers. In order to achieve this, we present the design of a three-step method that is required to go from Java source to a Promela model, the input language for the SPIN model checker.

There are a number libraries available for writing programs that require parallelization. Of these libraries, one standard has become the industry standard: MPI [6], or the Message Passing Interface. MPJ [4], short for Message Passing interface for Java, is an adaptation of MPI for the Java programming language. MPJ provides the same constructs as MPI, but in the form of a Java library. The Java platform is a popular platform for scalable computing and as such interest in MPJ is rising.

MPJ (as well as MPI) uses a paradigm called the message passing model in order to achieve parallelization. As it is a library for distributed computing, it considers a network of computers each executing the same program. Each of these computers is called a node. Each single executing node executes by itself, and when communication with other nodes is required, each node has a set of functions available for communication, supplied by MPJ. Among these are sending to specific nodes, receiving from any or all nodes and interprocess management such as barriers. To achieve this, MPJ and MPI implement a set of methods to use for each node involved in the communication. For example, send() and recv() methods are implemented on each client, that facilitate the sending and receiving of data amongst nodes, with the the methods only returning (also named blocking) until the operation has succeeded. In order for nodes to be identifiable with each other, each node is given a unique number, the rank, which can be used in communication.

MPJ applications are deployed in various critical processes, and each of these may encounter various errors that will prevent the program from completing properly. As mentioned in the last paragraph, communication methods may block execution of a program entirely. If a situation occurs so that a program never leaves such a blocked situation, we call this a safety property violation. Many safety property violations in multimode or multithreaded systems can be identified [3].

Tools are readily available for the verification of parallel algorithms: one may abstract the algorithm by hand to have it verified by a readily available model checker. These methods do not verify parallel programs, which are the units that should be verified in the end.

2. PROBLEM STATEMENT
Various problems can arise when this communication can occur. For example, a deadlock may occur in the case of the following program:

```java
public static void main(String[] args) {
    MPJ.init();
    Comm comm = MPJ.COMM_WORLD;
    // Send a message to node 0 if this node is 1
    if (comm.rank() == 1) {
        comm.send(a to 0);
    }
    // If this is not 1, send a message to node 1 else
    comm.recv(a from 1);
    MPJ.finish();
}
```

Figure 1. A simple Java example program

This (simple) example shows deadlock possibility. Because the MPJ recv() method blocks execution of the
problem until something is received, this program only finishes when this program has one or two nodes involved. Any node added beyond these two nodes does not complete its execution, as the node calls its blocking `recv()` function and never receives anything to complete it. The

```java
active proctype P(int rank, size) {
    if
    : rank == 1 -> send(to 0)
    : else -> recv(from 1)
    fi
}
```

This example shows some of the choices that have been made in order to generate code. First, we do not abstract the value of the rank of this current channel. We believe this can be adequately simulated. Values obtained from inputs other than MPJ will be abstracted. Secondly, a means to reimplement MPJ's communication functions must be found for Promela. Although Promela does facilitate means for interprocess communication, we present some Promela constructs that provide semantically equivalent constructs to MPJ method calls.

Following above considerations, we formulate the following research question and its subquestions:

**Main question:** How can one create a model from a MPJ-based Java program which we can use for model checking?

1. What information in the program is (and is not) useful to a model checker?
2. What control flow representation is the most useful for verification?
3. How can this control flow representation be processed to feed to a model checker?

These questions reflect the three steps that are necessary to get from a Java source file to model checker input.

## 3. EXISTING WORK

Although there have been, to our knowledge, no attempts been made to make a verification tool for MPJ, other work has been made in the verification parallel programs and Java programs.

### 3.1 Concurrent Java to model checker

The possibility of converting Java programs to the Promela language for verification using SPIN is an approach used in a tool called Java PathFinder [5]. Although Java PathFinder does not use Promela as a backend anymore, the concepts presented should still prove useful. This method focused on verification of multithreading concepts in Java. It could translate a Java source file using the standard Java multithreading constructs to Promela.

### 3.2 Control flow abstraction from Java

A paper by Alur et al. [2] describes a method for abstracting a means for correct ordering from Java classes. In essence, he uses predicate abstraction to filter what order methods can be called. Part of this process is making a control flow abstraction. The presented tool, JIST, is not available any more, but its methods are based upon the Soot framework which is used in this work for control flow abstraction as well.

### 3.3 CEGAR

In order to prevent a state explosion in the resulting control flow representation of the verification tool, variables that exist within the program must be abstracted. Each of the value ranges a variable may have is categorized among semantically equivalent values, so that the control flow may split only for the categories a variable value may have.

The generally preferred approach for this categorization is counter-example guided abstraction refinement (CEGAR), which works as follows. First, this method starts by ignoring all the effects of variable values on the control flow. If it encounters a scenario in which a safety property is violated, it checks whether the scenario can actually occur when the values of variables are taken in account. Then, the algorithm iteratively recurses until a violation of a safety property occurs or no violation occurs at all, which indicates whether the program contains violating...
3.4 Verification of MPI
Siegel et al. showed [9] a method to verify numerical parallel MPI programs. They define a numerical program to be a monolithic program that accepts a set of numbers as input, and produces a set of numbers as output. A second requirement was that a sequential program is given that is semantically equivalent to the parallel program.

Both requirements ensure that a program exists out of mostly simple arithmetic operations and that it gives the same output for the same input set. This subset of programs ensures a limit on the number of operations that need to be considered in control flow abstraction. The method requires an equivalent sequential program to be given. Predicates are then abstracted from the sequential program using symbolic execution, a top-down approach that uses substitution in arithmetic operations to obtain bounds on the values certain expressions can have. Using these predicates a model checker was used to verify whether the predicates still held in a parallel situation.

Siegel and Avrunin published [8] a method of verification for MPI functions that execute synchronously. Importantly, they show that many important properties for MPI can be verified when considering peer-to-peer communication between nodes, but not when broadcasts or receives from wildcard nodes are employed. Lastly Palmer et al. [7] have done extensive work on making formal semantics on 35 MPI-I functions, all of which have been adopted in MPJ [4]. These can be used to formulate what happens to the control flow when an MPI or MPJ call is made.

4. METHODOLOGY
The research questions posed offer a scaffold for the project: building a tool that provides this translation. The project will be divided in three parts, and a set of tools to be used will be selected. Due to time constraints, a full model checking toolchain is not feasible: instead, our target is to provide a toolchain that has all required elements, but can only verify a few simple Java programs.

4.1 Tools
We will make use of the two following tools.

- **Soot** A library intended for analysis of low-level Java operation, Soot [11] provides an intermediate language called Jimple which is a language similar to Java bytecode. This representation is helpful in extracting important expressions regarding the control flow.

- **SPIN** As the dominant tool used for verification, SPIN [6] will provide the model checking work on the abstracted control flow.

4.2 Information extraction
The first part we researched was information extraction: we must find a method that can abstract useful information from an MPJ program and, where possible, can ignore irrelevant information. We must determine which information is useful and which is not. It is expected, but not certain, that existing libraries can provide help in this.

A hurdle that must be overcome in information extraction is the fact that control flow abstraction in strictly literal sense leads to a state explosion. In order to obtain an acceptable size for the control flow graph, we must find a method to reduce the number of states by methods. CEGAR is an example of such a method, but unfortunately time restrictions held back its implementation in the tool.

4.3 Control flow abstraction
In the second part of the project, we use the information extracted in the last part, and transform it to a proper structure. An often found representation for this is a control flow graph. Furthermore, we will make a selection of the

4.4 Model checker language
This control flow representation needs to be fed to a model checker. Therefore, we must transform the abstracted control flow to a representation that SPIN can use for verification. The most straightforward way to do this is to generate Promela.

Promela is sufficient for representing algorithms in a parallel setting for many applications [8]. The MPJ library presents a complete inter-process messaging API. Additional work is done to make a reusable MPJ simulation in Promela.

We have combined these three steps in the a proof-of-concept tool, which we have named *Magggy*. The design of this tool shall be covered in the following section.

5. MAGGY

The design of *Magggy* is covered here. We will first give a brief overview, and subsequently explore how data is processed to generate model code from a Java source. We present an overview of the tools workings, and subsequently cover the translation of control flow and node communication. Finally, we will discuss the implementation of these techniques withing Maggy.

5.1 Overview

*Magggy* is a command line tool built on Java. It requires two configuration files: a Java class file compiled from a Java source that utilizes MPJ calls, and a configuration file that specifies how many processes should be simulated in the model. Using the techniques described further in this section, the tool obtains the control flow and communication from the class file. It then translates this information to a semantically equivalent Promela model.

5.2 Control flow

When a Java program continues execution at a point that is not necessarily one statement further down than the previous executed statement, we call this a control flow change. Java contains many statements that issue a control flow change. Within method bodies, the most often used are if, while and for-statements as well as the less often used goto and return statements. As these often signify code logic, we shall name these logic control flow change. Other control flow changes include method invocation and returning. Considering the application of MPJ calls within algorithms, we determined the scope of Maggy to support only the logic control flow changes.

Support for method calls is not implemented in Maggy, and the tool therefore only supports control flow in the Java `main()` method. The only method calls processed are MPJ calls for communication. These are not implemented through the parsing of the Java code, but by similar constructs in the Promela language.

Upon examination of the generated JVM bytecode and Jimple code, it was determined that these logic control
flow changes were determined by a combination of if- and goto-statements only. It was therefore determined that the information abstraction need only consider these statements to obtain a sufficient control flow representations. As a result, Maggy considers a semantically equivalent Jimple representation rather than the Java source files directly.

5.3 Communication
MPJ offers a large amount of communication methods [4]. These include blocking send and receive operations, non-blocking send and receive operations and several communication management tools. It was shown in previous research that blocking send and receive can be easily model checked, but extra work must be done to provide the same semantics for non-blocking MPJ operations Sieg07. It was therefore determined to focus on blocking operations.

Promela provides communication constructs [6] that share identical semantics with MPJ blocking send and receive. The Promela channel receive operator checks a Promela channel for a matching variable that is placed in the channel, and blocks further execution until one is found. This is the same behaviour as the MPJ blocking recv() call [7]. Similarly, the Promela send operator provides a method to input something in a communication channel which blocks when the channel is still occupied, in complete correspondence with the specification of the MPJ send() function [7].

A direct mapping could therefore be made between MPJ calls and Promela operators. The MPJ functions specify more functionality than the Promela operators. MPJ recv() calls can specify a specific other process to receive from. This is implemented in Promela by not sending a message by itself, but by encapsulating it in a data structure that also contains the source message. Promela provides conditional readouts of channels so it can be specified. A further feature that is implemented is wildcard sending and receiving - a Promela send() or receive() call may target all other channels. This was implemented by multiple send operations for the Promela send operator, and a wildcard receive for the Promela receive operator.

5.4 Implementation
The proof-of-concept relies on the Soot library for control flow abstraction. Maggy uses Soot to obtain a data structure representing the Jimple code, as shown in figure 3.

Maggy is a plugin within the Soot Jimple generation. Soot takes a Java class file as input, and processes this file iteratively. It first obtains a Jimple representation after using its internal representation (named BAF representation), which it then translates to a Jimple representation, and finally to a Jimple output file. Maggy plugs in to use Jimple's internal representation [11] as input, which comes in the form of an abstract syntax tree, and it outputs a Promela model file in addition to Soot's output.

The Soot internal Jimple representation consists of a list of classes named Units that correspond roughly to individual statements in Jimple code. This list of Units is transformed into a control flow graph by Soot's tools. This yields a graph of Units that splits or transitions to another location in the graph in case of a control flow change. Using a depth-first search method for traversing the Jimple syntax tree, each Unit in the graph is marked for its relevance.

A Unit is marked relevant or irrelevant based on its type. As it was determined that only Jimple if and goto statements were necessary to represent the control flow, it marks these units as relevant. Furthermore, statements that call the MPJ send and recv methods will also be marked as relevant. Finally, Maggy will also provide a method to provide symbol names that it must not ignore. By default, Maggy ignores statements that involve variables as the many different values a variable might have may lead to state space explosion [9].

5.4.1 Architecture
This representation is processed by Maggy using the following components:

![Figure 3. Maggy’s placement into the Soot process. Arrows represent Soot’s internal translation steps.](image)

![Figure 4. Overview of Maggy’s architecture. An arrow represents flow of data.](image)

The Analyzer is where Maggy receives the Jimple representation from Soot. It calls the CFGBuilder, which uses Soot’s libraries to build the control flow graph. After that has succeeded, it will walk the control flow graph (using depth-first search) to mark Units relevant or irrelevant based on what filters it has defined. The control flow graph begins being marked irrelevant completely. A filter takes a control flow graph, and outputs a list of Units...
that it deems relevant. Maggy can be extended by further filters that mark more of the program relevant.

5.4.2 Backend
The control flow graph and the list of marked units is subsequently passed into a backend. Maggy has two defined backends: a text backend (for debugging purposes) and a Promela backend. Each backend issues a depth-first search of the control flow graph, only considering the statements marked as relevant. For each statement, it finds a corresponding output call, and writes that to an output file.

The Promela backend makes a one-to-one mapping of control flow statements from Jimple to Promela. Again, only the if and goto statements from Jimple are used, and each of those has a corresponding Promela counterpart, which is used directly. if-statements are left undetermined: as by default variables are not taken into account when generating the Promela, we use a Promela construct that determines the execution of the conditional statement in the if-body to be determined randomly by the model checker. The representation of control flow and communication is put in a standard template for the Promela code, which completes the model by defining the proper communication channels. In section 6 we will cover further details on the Promela output.

Some information for model checking is part of the MPJ configuration, and not part of the Java files. Most importantly, MPJ needs to be supplied the number of processes as well as the number of concurrent processes it must run (simulating that the Java expression in the if-statement is evaluated to true) or to continue execution immediately after the if-statement (simulating the expression evaluating to false). The first option is implemented by using a goto-statement to jump to another part of the model code. The second option is implemented by issuing a Promela else statement that executes nothing, after which it will continue execution after the if-statement. SPIN will choose indeterministically between these two options.

We illustrate the above in the following example.

```java
if(condition) {
    a;
}

b;
```

Here, a and b refer to a collection of other code statements.

```java
if :: goto label0;
:: else ->;
fi;
```

Figure 5. Promela output for the Simple communication program

From this figure, we may observe a number of things about the generated Promela model. We will discuss the Maggy translation details in two parts.

6.1 Template

Unlike the Java program, the Promela program requires definitions on communication channels (whereas in Java, communication channels are provided implicitly by MPJ) as well as the number of concurrent processes it must run (with MPJ, this is based on how many processes a user starts). Maggy uses the Promela init initialization section to define communication channels for each process. There are as many communication channels as there are processes: they are defined as an array with each element of an array the channel a node may check for messages it has received. As this is makes a good analogy with concept of mail, the array of communication channels is named postbox. The data structure used for the channels is explained in section 6.1.3.

6.1.2 If and goto

if statements encountered in the Jimple program are translated to corresponding if statements in Promela syntax. Each Promela if statement is given two options: to execute the statements that are enclosed in the if statement (simulating that the Java expression in the if-statement evaluated to true) or to continue execution immediately after the if-statement (simulating the expression evaluating to false). The first option is implemented by using a goto-statement to jump to another part of the model code. The second option is implemented by issuing a Promela else statement that executes nothing, after which it will continue execution after the if-statement. SPIN will choose indeterministically between these two options.

We illustrate the above in the following example.

```java
if(condition) {
    a;
}

b;
```

Here, a and b refer to a collection of other code statements.

```java
if :: goto label0;
:: else ->;
fi;
```
goto label1;
lbl0:
a;
lbl1:
The Promela program executes the a collection of statements only if the if statement evaluated to true. This same structure can be seen in figure 5.

6.1.3 Communication
Communication can be seen to be implemented using the Promela send and receive operators. The channels contain a data structure featuring two integers. The first integer signifies which node has sent the message in the channel. The second integer is a dummy value that models data sent. When a node sends data it provides its own rank in the data structure. A broadcast is implemented by repeating the send operation in the Promela model for each node. When a node receives data, it checks whether the node sender has received something by specifying which rank it needs to encounter in the channel. If a node is specified to receive from any node it uses the Promela wildcard operator '\_'.

6.2 Naive coprime
The naive coprime algorithm generates a list of numbers that are coprime with a certain number. The computation of a single number is done by a single node, and each node sends its number to node 0, which outputs a result.

The Java source code can be found in the appendix. A hand-made control flow graph is given below. The Promela code, as well as the Java source code, can be mapped unto this control flow graph.

```
proctype main(int rank) {
    if :: goto label0;
    :: else -> ;
    fi
    goto label1;
    label1:
    if :: goto label2;
    :: else -> ;
    fi
    goto label3;
    label3:
    label2:
    postbox[rank]?{_,1} ;
    if :: goto label1;
    :: else -> ;
    fi
    label0:
    if :: goto label4;
    :: else -> ;
    fi
    goto label5;
    label5:
    postbox[0] ! { rank, 1 };
    label4:
}
```

The outputted model bears little resemblance to the original Java program, as it is stripped from all variables and its helper method. With respect to control flow of the communication however, it is semantically equivalent. Some correspondence with the original Java program is gone through the depth-first search nature of the control flow graph parsing algorithm.

7. CONCLUSION
In this paper, we have demonstrated that it is possible in principle to create semantically equivalent software verification models from MPJ-using Java programs. To show this, we created a tool named Maggy, which abstracts control flow and communication calls from Java class files and creates a Promela model out of them.

The models that Maggy creates are a faithful representation of the control flow and communication calls from the original program. Due to Maggy’s omittance of variable names however, the control flow is not governed by the values of variables and therefore undeterministic. Sadly, this means that Maggy’s results are not suitable for verification by SPIN in its current state, leaving this open for future work.

It was determined that these features of the original Java program are important for model checking:

- **Control flow** Java code has several constructs that modify the order in which communication calls are made. To obtain a proper model from this code, we must at least obtain logic control flow changes, defined by being represented by if and goto-statements in the bytecode, or more.

- **Communication** We must also abstract the communication that occurs in between the nodes, whether it is a send or receive, and to or from which node this communication must occur.

Figure 6. The control flow graph for the naive coprime program

```
chan postbox[2] = [0] of { int, int };
init {
    run main(0);  run main(1);
}
```
The preferred representation of this information is the control flow graph with nodes representing individual statements. Its advantages include easy generation and its suitability to be processed in the model generation. This control flow graph is accompanied by a marker list, which categorizes which nodes should be used in model generation.

The control flow graph was processed using a depth-first search algorithm that processed each branch in the control flow. It was possible to make direct translation of each item in the control flow graph as each node of the graph represented a statement. The representation of nodes as states is preferred as it allows each of the states to be classified for its importance individually based on its type. This classification makes it possible for each node to be ignored when it is not needed in model generation.

Due to each node corresponding to an individual statement, it made possible for there to be a one-to-one mapping of statement to model code, wherever such statements were determined to be relevant. It was therefore possible to use a simple translation while using depth-first search to traverse the control flow graph to convert it to model code.

8. FUTURE WORK
Maggy was intended as a proof-of-concept and is limited in nature as well as scope. Although translation of the Java source code is completed successfully not enough testing time was available, which we recommend with the highest priority. Furthermore, we advise research in the following areas.

Maggy has several limitations (see section 5.5) which can be improved upon. First and foremost is the implementation of an automatic method for the model to include variables [9], which remains unexplored territory for MPJ programs. Other improvements may include an automatic CEGAR algorithm that automatically traces back variables if their absence is the cause of the model verification failure. Maggy provides the foundations for this, but does not implement it.

This project was accomplished using several established tools. It is however possible to create an in-situ verification tool for this [10, 5]. Although we can recommend following the same methods as presented in this paper, an in-situ tool which integrates program parsing reduces the overhead of having a general-purpose parser, and the limitations of having a general-purpose model checker.

Promela channels are not sufficient for a full simulation of MPJ calls. Using Promela channels, one can only implement simple read- and write operations. For simulation of specific buffer sizes, flexible communication channels or constructs like barriers, one must implement a Promela construct that is not provided natively. Further research is necessary to provide adequate modeling for all MPI/MPJ communication constructs, although some research has been ongoing [8].

9. REFERENCES

10. ACKNOWLEDGEMENTS
Many thanks to Michael Weber, who oversaw my progress when doing this research. His insights, hints and remarks were very valuable for my progress in this project. Thanks also to my fellow referaat track members, whose weekly collaboration and feedback continually improved this project and made this first step in academia very educational and pleasant.
APPENDIX

The Java source code for the examples mentioned in the paper is presented here.

A. JAVA SOURCE FOR THE SIMPLECOMMUNICATION PROGRAM

```java
import ibis.mpj.Comm;
import ibis.mpj.MPJ;
import ibis.mpj.MPJException;

public class Main {
    public static void main(String[] args)
        throws MPJException {
        MPJ.init(args);
        Comm comm = MPJ.COMM_WORLD;
        int tag = 1;
        Integer[] data = new Integer[1];
        if(comm.rank() == 1) {
            data[1] = 0;
            // Send data to 0
            comm.send(data, 0, 1, MPJ.INT, 0, tag);
        } else {
            // Receive data from 1
            comm.recv(data, 0, 1, MPJ.INT, 1, tag);
        }
        MPJ.finish();
    }
}
```

B. JAVA SOURCE FOR THE NAÏVE COMPRIMES PROGRAM

```java
import ibis.mpj.Comm;
import ibis.mpj.MPJ;
import ibis.mpj.MPJException;
import java.util.HashSet;
import java.util.Set;

public class Main {
    public static void main(String[] args)
        throws MPJException {
        MPJ.init(args);
        Comm comm = MPJ.COMM_WORLD;
        int tag = 31415;
        // Number to be checked size is determined
        // by number of nodes
        // Node 0 collects from all nodes
        if(comm.rank() == 0) {
            Set<Integer> resultSet = new HashSet<Integer>();
            int numLoops = comm.rank();
            while(numLoops-- >= 0) {
                int recv_buf[] = {0} ;
                comm.recv(recv_buf, 0, 1,
                        MPJ.INT, MPJ.ANY_SOURCE, tag);
                int value = recv_buf[0];
                if(value != -1) // if not invalid
                    resultSet.add(value);
            }
            System.out.println("The comprimes of " + comm.size() + ")
        } else {
            int result;
            int gcd = Main.gcd(comm.size(), comm.rank());
            if(gcd != 1)
                result = -1;
            else
                result = comm.rank();
            int send_buf[] = { result };
            comm.send(send_buf, 0, 1, MPJ.INT, 0, tag);
        }
        MPJ.finish();
    }
    
    private static int gcd(int a, int b) {
        if(b == 0) {
            return a;
        } else {
            return Main.gcd(b, a % b);
        }
    }
}
```