Automatically testing semantics preservation in SLCO to Java transformation

Wiłkowski, M.

Award date:
2019

Eindhoven University of Technology

Master Thesis

Automatically testing semantics preservation in SLCO to Java transformation

Maciej Wiłkowski

supervised by
Dr. Ing. Anton Wijs

November 8, 2018

Abstract

The thesis examines the problem of functional property preservation in model to code transformations between SLCO models and Java source code. We describe why it is important to ensure that the executable code generated from a model preserves the properties of the original and how it is related to the Model Driven Development methodology.

Semantics preservation is a property of the transformation that guarantees that two systems have the same potential behavior. We explain how, by proving that the translation is semantics preserving, we obtain a guarantee that the translated code will preserve the functional properties of the model.

We develop a technique based on a graphical representation of the program execution, *Control Flow Graph*. We check whether such a graph describing the semantics of an SLCO model is equivalent to a graph describing the semantics of the corresponding code by means of bisimulation checking. We argue that this approach provides an automatic way to check the preservation of semantics.

We test the concept by applying it first on the original Java translation of an SLCO model and then on a translation modified not to preserve the functional properties of the original.

We conclude that the method effectively determines functional property preservation between the model and the generated source and can be easily incorporated into the development process.

Contents

Chapter 1

Introduction

1.1 Background

The growing complexity of systems has created a need for more robust error-proof development methods. It has been estimated that each 1000 lines of delivered code will contain 15-20 errors[20]. This can be troublesome when we have to rely on the system for safety and security, especially in areas like aeronautics or banking. An overlooked minor bug can result in putting lives at risk or in a multi-million dollar loss for the company. To minimize the risk the development process has to become more organized. Extensive planning, documentation and requirement analysis should be conducted before any actual programming happens. Unfortunately, very often the artifacts of this process become obsolete and fail to get updated if any changes occur at the development stage.

To solve this a Model Driven Engineering approach can be used which puts those artifacts at the center of the development process [8]. By using tools such as Domain Specific Languages, a problem can be expressed in a clear and succinct manner. The underlying technical details are often generic enough to re-use and can be modularized and tested separately. Furthermore, by abstracting them away we can make the tool accessible to people who possess domain knowledge of the system under development but not necessarily the technical knowledge of the system under development. Most importantly however, by simplifying the system into a model we can reason about in an organized way. We can specify desired properties of the system using propositional logic and using model checking[6] evaluate if they hold in our model. This enables us to formally verify that the desired property will always hold in our system. As

a result we can essentially make sure that our model is bug-free to the extent specified in the requirements.

Model Driven Engineering (MDE) aims to shift the development from modifying the source code to modifying the model. From there, by using model transformation we can transform the simplified system into a real-life environment. By having a full translation of the model into executable code, we remove some of the responsibility from humans and give them to machines. Ideally, the translation reuses code modules written in advance for a given scenario that have been thoroughly tested beforehand. Any additional change to the system can be reduced to a model change which is again verified and translated to working code. Those changes are smaller and easier to manage than the same changes in source code. As a consequence, we can eliminate regression bugs without writing a single test case. Another problem when developing complex systems is that the documentation tends to deteriorate with time. From the design document to the final product, a lot of assumptions lose importance and some new ones are getting created. Since development in MDE centers around requirements and a model (which in itself is a design document), to make a functional change to the system we need to update those artifacts as well. In effect we are forced to always synchronize the documentation to the end product.

Model Driven Engineering has proved successful in an industrial setting. In [3] authors describe how they have implemented MDE in their development process. It has been stated that due to usage of the technique a 33% reduction in test effort has been achieved. Furthermore in some components 65% - 85% of code is automatically generated, leading to significant reduction in workload and quality improvement. Most importantly it has been noted that it is not unusual to see a 30X - 70X reduction in time needed to correctly fix a bug. The observed improvement is attributed to the model being an illustrative part of the documentation, being able to fix the functionality at the model level and re-running the regression test suite on both model and generated code. Unfortunately the experience was not without any issues. The authors cited poor performance of the tools and generated code. The tool ecosystem was also lacking and immature, with no standardization in place. Finally there has been a noticeable problem in adoption due to engineers not having a lot of experience with the new methodology. Since no well defined MDE process was in place the adoption proceeded in a trial and error fashion. In some cases a need for performance improvement resulted in moving low level application code to the model layer, which resulted in the same problems as in hand code.

1.2 Problem statement

Most of the currently used model transformation languages are based on the Eclipse Modeling Framework[28], which defines Ecore meta-model as a way of specifying the models. Ecore is defined in terms of itself and is useful as it is a de facto standard when using MDE in a Java language environment. From Ecore specification EMF generally provides two approaches to transforming the models - rule based and graphical. In the first, implemented by ATL, a series of rules are defined. Those rules, executed one after another, bring the original model to desired form. The rules are transformation descriptions taking as an input elements from the original model and generating an output based on its type and attributes. The rules can be more complicated including iterators, filters and common set operators. The second approach makes the process of transformation more easy to use. In languages such as Henshin, by graphically laying out input and output metamodels, a mapping between the corresponding elements can be created. Unmapped elements are simply removed, while some advanced mappings can also be used on par with the features provided by the rule based approach.

Very often, the EMF infrastructure does not match the requirements of the project and internal tools not tied to the EMF infrastructure are used. A simple templating engine can serve the purpose in a highly customized environment[26]. By filling in stubs with pieces of code that are generated based on the model elements, a fully working application can be created.

The main problem with the mentioned solutions is that they do not provide any assurance that the correctness is preserved after the model transformation. There are multiple ways to check the correctness of the source model. One can use model checking to verify that the specification is never broken in the model. This however does not include the generation of source code. The output of such transformation may have different semantics and therefore not strictly adhere to the requirements.

To further complicate the matter, transformation languages do not have formal a semantics which makes it very hard to reason about them. There has been a couple of attempts at defining those semantics, such as [4] which formalize them by algebraic graph transformation. Most of them focus however on verifying transformation properties like determinism and termination. While it is useful to know if a transformation will always terminate and always produce the same output, it does not help to make sure that the result model is functionally correct. What makes the problem more difficult, very often the

end product is at a different abstraction level than the source. Models tend to focus on the structure, leaving the detailed behaviour to the generated code. To give an example, when specifying a statemachine model, we do not specify how the change of states happens, how we keep track of the current state and ensure thread-safe access of the variables. This information is added during the translation phase and therefore not checked during the verification of the model.

With aforementioned problems of undefined semantics of transformation languages and different levels of abstraction of transformed models, it seems that verifying whether transformations fully preserve model semantics might be an unresolved problem in the foreseeable future. While it may be difficult to formally verify the transformation language in general, another approach is to test a specific instance of a transformation where source and result models are already present. This thesis is trying to utilize commonly known techniques of control flow graphs and bisimulation to compare the models in terms of behaviour. By bringing both parts of the transformation to a common abstraction level we can deduce if the intended behaviour is preserved and therefore test if the end model still adheres to the specification.

1.3 Research Questions

To better understand how to solve the problem defined in the previous section, we formulate the main research question as follows:

RQ: *How can you verify that the generated code is correct with regards to the source model?*

We then split this general question into three more specific research questions, that will be addressed in this thesis.

The primary problem when designing software using MDE approach, is to prove that the software will adhere to the specification. While it is possible currently to reliably check the properties on the model, that does not infer that the code generated from the model will retain those properties. This leads to the following research question:

RQ_1: *How can you establish that the end product of model transformation preserves functional properties of the input?*

This however uncovers a different, connected problem with regards to abstraction level of the input and output models. It may be that the language constructs are not fully translatable and some structure might be impossible to reflect in one of the endpoints. This semantic gap might make the verification difficult and even impossible, which we will examine in the next research

question:

RQ_2: *How can you bridge the semantic gap between the transformed languages?*

To make the verification process useful for the common user, it cannot be too complex and difficult to conduct. To make the check suitable for the MDE workflow it has to be accessible even to people without strict academic background. This leads to the last research question:

RQ_3: *How can we automate the checking process, so it can be done with minimal user effort?*

Chapter 2

Preliminaries

2.1 Simple Language of Communicating Objects

Simple Language of Communicating Objects or SLCO[9] has been created at Eindhoven University of Technology. During that time ASML, the world's leading manufacturer of litography systems for semiconductor industry, was introducing UML diagrams as means of modelling systems in a model-driven software engineering process. The company experimented with transforming UML models to Parallel Object Oriented Specification Language (POOSL)[30] for performance analysis.

While developing a transformation framework for UML to POOSL models, a number of problems became apparent. UML lacks formal semantics and the diagrams are not necessarily complete or consistent. When working with bigger projects, using graphical tools can become cumbersome and the resulting diagrams difficult to read. Furthermore UML did not offer appropriate abstractions for the desired performance model and a number of constructs did not have their equivalents in POOSL. To solve these problems SLCO:

- Provides both textual and graphical environment.
- Provides same or transferable abstractions as a model checker input language.
- Uses model transformations as model refinements to close the gap between the target language semantics.
- Ensures both unrefined and refined models can be translated and verified for correctness.

2.1.1 Metamodel

An SLCO model consists of a number of classes, instances of those classes as objects and multiple channels through which these objects can communicate. Each class describes a behaviour of the component similarly to what can be expected in an objected oriented programming language. Each class consists of a list of statemachines, which describe the flow of the execution, a list of global variables that can be accessed within all statemachines and ports through which of the instances of the class can communicate.

A statemachine describes all possible states that can occur along with the transitions between them, furthermore each statemachine can define its own local variables that can be accessed only within the statemachine itself.

A transition is defined as being from one state (source) to another (target) and can optionally describe a series of statements that occur when taking that transition. They can optionally specify a priority, which determines the order in which the enabled transitions should be taken.

A statement describes a change of state or control flow in the statemachine. There are seven types of statement in SLCO language:

- **Assignment** changes the value assigned to a local or global variable.
- **Expression** guards execution of a transition.
- **Composite** is a block of assignments optionally guarded by an expression. Composite statement ensures that the expression and that the assignments are all performed in one atomic step.
- **Send Signal** sends a signal through the channel.
- **Receive Signal** receives a signal through the channel.
- **Delay** halts the execution for a specified time.
- **User Defined Action** an abstract action defined by the user.

SLCO also provides four types of channels: synchronous, asynchronous, lossy and asynchronous lossless, of which each can be either unidirectional or bidirectional. Each direction of an asynchronous channel is associated with a buffer. Signals can be sent through a lossless channel only if the associated buffer is not empty. Signals can always be sent through the lossy channels. In case the buffer is full, its contents will be replaced by the new signal. A metamodel of basic constructs in SLCO 1.0 is presented in figure 2.1[9]

2.1.2 Formal Verification of SLCO Models

SLCO models support formal verification of functional properties as described in[24]. SLCO framework provides a transformation scheme to the mCRL2[7]

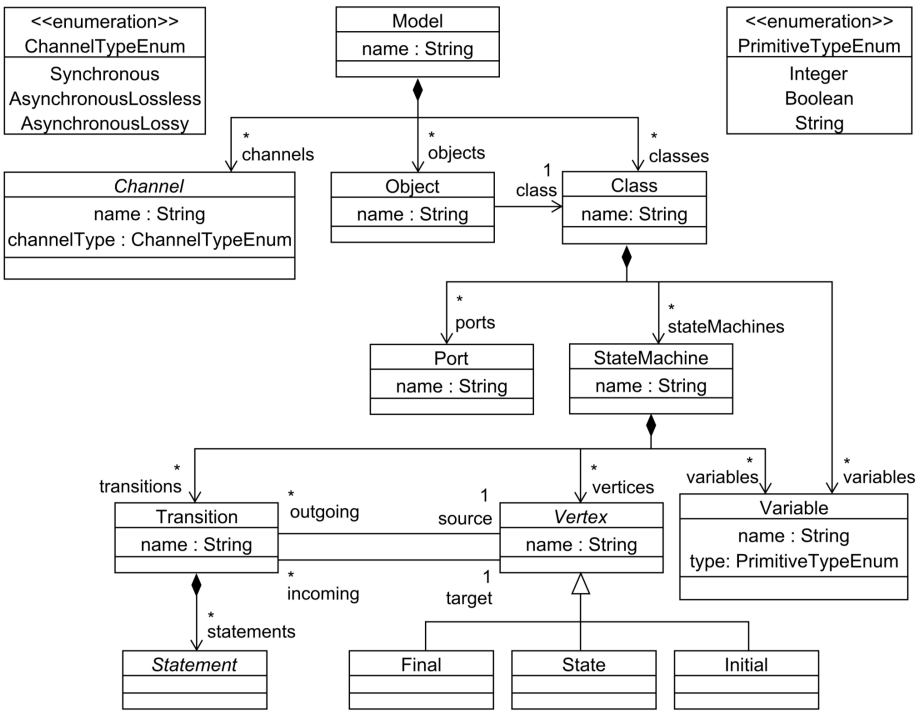


Figure 2.1: SLCO 1.0 Basic Constructs

process algebra. The desired properties are defined in terms of μ-calculus formulas. Finally both the properties and the model are combined into a Parametrized Boolean Equation System, the state space of which is then checked for compliance.

2.1.3 SLCO Model Transformations

SLCO language was designed with model transformations in mind. It currently supports exogenous transformations to aforementioned mCRL2 models, to Graphviz DOT visualization format and to Java source code to produce executable programs. Figure 2.2 presents possible capabilities of the system.

SLCO also provides a way of refining the model by endogenous transformations, some of which might include rewrite of the statemachines or replacement of user-defined actions with concrete behaviour. Some of the refinements can be verified for property preservation using the REFINER[32][25] tool. In other cases properties can still be checked by transforming the model to mCRL2 and verifying it.

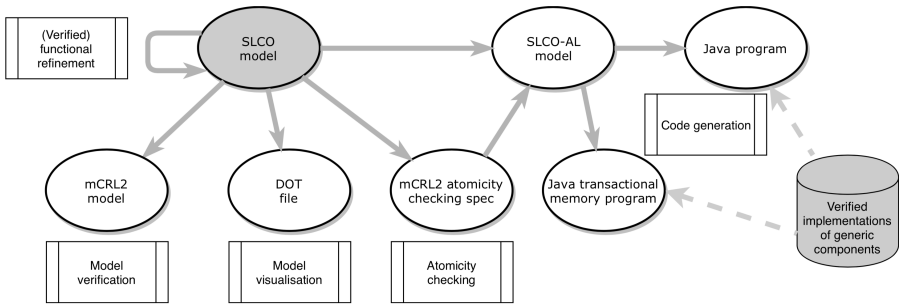


Figure 2.2: SLCO Framework Overview

2.2 SLCO 2.0 to Java Translation Framework

SLCO 2.0[24] introduces a new template based Java source code generation engine. The generation works by first parsing an SLCO text file and storing the model structure in memory. A generic template file is created, which is then read by the program filling it with processed parts of the model where applicable.

2.2.1 Model

A Model is translated to a main class of the resulting Java program. It also defines a class called `java_Keeper`, which is responsible for acquiring and releasing variable locks in the system. The model source defines and instantiates all global variables. All the states in the model will be translated to an enum contained in `java_State`. Furthermore the main class defines a main function, which will instantiate all the objects and start all the statemachines.

A translation of an example model is shown on listings 2.1 and 2.2. The listings contain the declaration of the `java_State` enum, which remains empty as no statemachines are defined. The listings also show the declaration of the `Keeper` class.

2.2.2 Class

SLCO class is not directly translated to Java class. Instead each statemachine is translated to a separate subclass of a thread class. This permits all the statemachines to run independently, except when there occurs an access conflict with relation to global variables. Each of those translated statemachines holds a

Listing 2.1: SLCO Model

```
model Example {
  classes
    A {
      state machines
    }
  objects
    a : A()
}
```

Listing 2.2: Java Model

```
// main class
public class Example {
  // The threads

  // Enum type for state machine states
  public enum java_State {
  }

  // Global variables

        // Lock class to handle locks of global variables
        class java_Keeper {...}

        // Constructor for main class
        Example() {
                // Instantiate global variables
                Example.java_Keeper java_k = new Example.java_Keeper();
        }
  ...
        // Run application
        public static void main(String args[]) {
    Example java_ap = new Example();
    java_ap.startThreads();
    java_ap.joinThreads();
        }
}
```

current state variable, which stores the state active at the moment. Transitions are modeled as an infinite while loop, which checks the current state and executes statements associated with it. In case there are two transitions coming from the same state, the system has to make a nondeterministic choice. In this case a random number smaller than the number of outgoing transitions is generated, which enforces a choice of a particular transition.

To visualize the translation we present a listing of an SLCO model (listing 2.3) together with the part of the code generated from the `A` class (listing 2.4). It is visible that no direct translation of the class exists. Instead the `T` statemachine is translated to a `java_TThread` class, which inherits from the `Thread` class of Java. The statemachine variable initializations are translated into class properties and initialized in the constructor. The transitions are translated inside the `exec` method of the class. This method defines the main loop of the statemachine along with the switch checking the value of the current state and redirecting the flow to the case responsible for handling the transition.

2.2.3 Channel

The SLCO 2.0 framework does not yet implement the channel construct. It is very likely it will follow the implementation described in[33]. The generated Java code essentially leverages `ArrayBlockingQueue` and `SynchronousBlockingQueue` classes. In case of an asynchronous channel, a defined channel class is a wrap-

Listing 2.3: SLCO Class

```
classes
  A {
    variables Boolean i := True
    state machines
     T {
      variables Integer j Boolean k:=False
      initial T1 states T2 T3
      transitions
        from T1 to T2 {}
        from T1 to T2 {}
        from T2 to T1 {}
     }
  }
```

Listing 2.4: Java Class

```
...
  class java_TThread extends Thread {
                private Thread java_t;
                private String java_threadName = "TThread";
                // Current state
                private Example.java_State java_currentState;
                // Random number generator to handle non-determinism
                private Random java_randomGenerator;
                // Keeper of global variables
                private Example.java_Keeper java_kp;
                // Array to store IDs of locks to be acquired
                private int[] java_lockIDs;
                // Thread local variables
                private int j;
                private boolean k;

                // Constructor
                java_TThread (Example.java_Keeper java_k) {
                        java_randomGenerator = new Random();
                        java_currentState = Example.java_State.T1;
    java_kp = java_k;
    java_lockIDs = new int[0];
                        j = 0;
                        k = False;
                }

                // Execute method
                public void exec() {
                        // variable to store non-deterministic choices
                        int java_choice;
                        while(true) {
                                switch(java_currentState) {
                                        case T1:
                                                java_choice =
                                                ↪  java_randomGenerator.nextInt(2);
                                                switch(java_choice) {
                                                        case 0:
                                  java_currentState = Example.java_State.T2;
                                                                break;
                                                        case 1:
                                java_currentState = Example.java_State.T2;
                                                                break;
                                                }
                                                break;
                                        case T2:
                                                java_currentState =
                                                ↪  Example.java_State.T1;
                                                break;
                                        default:
                                                return;
                                }
                        }
                }
...
```

per around the `ArrayBlockingQueue` object. The synchronous channel replaces the communication by `ArrayBlockingQueue` with `SynchronousBlockingQueue` but still retains the former as means of checking the signal before starting the communication.

2.2.4 Statements

The 7 types of SLCO statements are translated to Java code in the following manner:

Assignment statement

Assignment statement having a direct equivalent in Java, will translate to variable assignment.

Expression Statement

Expression statement translates to a conditional statement, which will break back to the main loop if the expression is a first statement. Alternatively if the expression is not a first statement in the transition, the expression will block the transition until the condition evaluates to true.

Composite Statement

Composite statements will translate to Java code in the same manner as expressions and assignments. A special class has been written to preserve the atomicity of the composite statement. The locking mechanism of the class will ensure that it is impossible to interrupt the execution of the statement. It will also lock the global variables to enforce exclusive use.

Send Signal & Receive Signal Statements

The send and receive signal statements depend highly on type of channel that is used. They are therefore abstracted into a simple `send` or `receive` method calls on a specific channel object. This is however not yet implemented in the version 2 of the translation framework.

Delay Statement

The delay statement translation is not yet implemented in SLCO2.0. To model the passing of time in a statemachine, the previous version of the framework will first record the time at the start of the statemachine in a variable `start`. The delay statement is then translated as a conditional check whether the difference between current time measured using `System.currentTimeMillis` and the `start` variable has surpassed the specified delay time.

User Defined Action

User defined actions are not yet implemented in the translation framework. Since the actions do not have any precise semantics, it might be problematic to translate them into meaningful Java code.

2.2.5 Atomicity

A significant problem when constructing a model to source transformation of a concurrent system is to ensure the preservation of atomicity between the model and the implementation. Due to non-deterministic potential interaction between threads, accessing shared variables can give rise to data races, which in turn may lead to undesirable behaviour. SLCO semantics provide a strong notion of atomicity - each statement is atomic. What it means, is that each statement behaves as if it were the only computation performed at any given moment. Effectively, no statemachine can have an influence on any other during execution of a statement.

In [33] author uses a concept of serializability to prevent any erroneous interaction between the threads in SLCO to source translations. To prevent interference, the strongest assurance would come from never running more than one thread concurrently. This is however impractical in most setups. Serializability is a concept that weakens that principle by ensuring that only code that does not affect global state can be run concurrently. In a Java program resulting from an SLCO model translation, the global state is represented by a number of shared variables accessible to many threads. It must then be ensured that the code executed by a thread will not access the same shared variables as code running on any other thread concurrently. SLCO translation uses serializability by defining a Java wrapper class over shared variables. By using lock and unlock operations of the class, the thread ensures that it will have exclusive access to those variables. A well known problem involved in acquiring locks on variables is a lock-deadlock. In a situation where there exists a circular dependency on the variables between the threads, it may occur that some threads will try to compete over a set of locks without releasing their own first. As a result no threads will be able to acquire locks and the system will not continue with the execution.

The issue of lock-deadlock is solved using the notion of ordered locking. Lock ordering ensures deadlock freedom by enforcing that the variable locks will be requested only in a specific order. For example, two threads $T1$ and $T2$ are accessing same variable locks A and B. Suppose $T1$ wants to access locks A

and B in that order and $T2$ in the reverse order B and A. Without the ordered locking $T1$ would first obtain the A lock and $T2$ the B lock. In this scenario, a deadlock will occur as $T1$ now waits for the release of the B lock held by $T2$ and $T2$ waits for release of the A lock held by $T1$. To solve the issue, by enforcing that $T2$ will have to access the lock in order - namely A before B, the risk of deadlock is eliminated as $T1$ has to first release ownership of A before $T2$ can obtain B. The downside of the method is that as all locks acquired by each thread needs to be known beforehand. Because we have full information on which variable locks will be accessed in an SLCO transition, we can use ordered locking to prevent deadlocks.

The thesis of Zhang [33] further verifies using a the code verifier tool Verifast[27], that by using both serializability and lock ordering the atomicity specification is preserved between the model and Java translation. Since the atomicity property preservation has been verified in [33] for all possible SLCO to Java translations, this releases the need to prove it separately on a model basis.

2.3 Control Flow Graphs

Control Flow Graph [2] is a data structure used to represent the control flow of the program. It is commonly used whenever an analysis of how a program executes is needed. An implementation of CFG is commonly found in compilers and other tools that need to codify the relationships between the instructions to optimize the runtime. It is also used in reachability analysis to track and remove the code that will never be executed.

A Control Flow Graph is a directed graph where each node represents a basic block and the edges represent a path in which those blocks can be reached. A basic block is a sequence (sometimes containing only one element) of instructions containing only one entry point and one exit point. What it means is that all the information about the program flow is represented by connections of the nodes instead of by the nodes themselves. A directed graph G is an ordered pair $G = (B, E)$ where B represents a set of nodes (blocks) and E represents a set of directed edges.

A Control Flow Graph is similar to a State Diagram in being a directed graph with edges representing a sequence of actions occurring in a system. It is different however in the fact that a node does not contain any information about the state of the variables or the execution stack. Each node is merely a change to the predefined starting state, which makes the graph more concise but also makes it impossible to argue about the system state in isolation.

Every node in CFG represents a step in computation, which may have a different meaning in various programming languages used as precursors for the graph. Some operate directly on registers and memory addresses, while some abstract those details out to variables and functions. It is possible to model a Java program in a CFG form. It is also possible however to focus on runtime intermediate language or even assembly code, which will be executed by the processor. The choice of level of abstraction is driven by the type of analysis needed as well as the granularity and precision required.

An example of a CFG generated from a Java function is presented in figure 2.3. On the graph we can observe the nodes that will alter the flow of the program based on a certain condition. We can also observe the nodes that will change the values stored in variables. The nodes are connected with edges representing the result to which the condition expression evaluates to. Alternatively the nodes are connected with fallthrough edges that represent the direct flow as specified by the order of instructions.

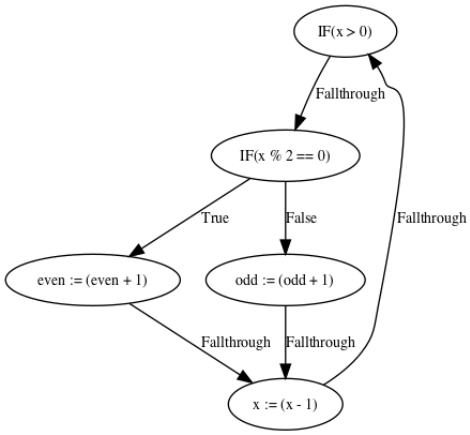


Figure 2.3: Example Control Flow Graph of a Java Function

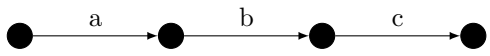
2.4 Labelled Transition Systems

When developing a complex system, designers need a formal way of analyzing the system behavior. This can be achieved by ordering the actions that can be taken at different points of a program into a Labelled Transition System (LTS)[12].
An LTS is defined as a five tuple $A = (S, Act, \rightarrow, s, T)$ where:

- S - a set of states possible in the system.
- Act - a set of actions that might be taken in the system.
- $\rightarrow$ - a set transitions between states $S \times Act \times S$
- s - the initial state.
- T - a set of terminating states.

In a labelled transition system a state in which a program can be is connected by one or more actions with different states, which are results of those actions. In case there is more than one action with the same label coming out of a state, we talk about nondeterminism. A nondeterministic state allows to model behaviour even when the exact behaviour is not clear. A state, which is not part of T and has no outgoing transitions is said to cause a deadlock in the system. This is usually an unwanted behaviour as the system has not completed successfully and is unable to continue.

Figure 2.4: Example LTS



2.5 Bisimulation Relation

Having a structure that captures the system behaviour, a formal way of capturing whether two systems are equal in behaviour is needed. This can be accomplished by the bisimulation relation[12]. If two systems specified in terms of LTS are bisimilar to each other they cannot be distinguished by any form of behavioral observation. For all intents and purposes these systems can be thought as equal in terms of behavior.

Given two labelled transition systems $A_1 = (S_1, Act, \rightarrow_1, s_1, T_1)$ and $A_2 = (S_2, Act, \rightarrow_2, s_2, T_2)$, we say a relation R is a (strong) bisimulation if and only if for all states $s \in S_1$ and $t \in S_2$ such that sRt holds, it also holds for all the actions $a \in Act$ that:

1. if $s \xrightarrow{a}_1 s'$, then there is a $t' \in S_2$ such that $t \xrightarrow{a}_2 t'$ with $s'Rt'$.
2. if $t \xrightarrow{a}_2 t'$, then there is a $s' \in S_1$ such that $s \xrightarrow{a}_1 s'$ with $s'Rt'$.
3. $s \in T_1$ if and only if $t \in T_2$

Two states s and t are bisimilar if there is a bisimulation relation R such that sRt. Two labelled transition systems A_1 and A_2 are bisimilar if and only if their initial states s_1 and s_2 are bisimilar.

In plain English description, two states are related by bisimulation if any

action that can be done in one state can be done in the other state as well. The actions of the second state simulate the first.

Figure 2.5 shows an example of a bisimulation relation (represented with a dashed line) between two systems. It may be surprising to see that the s_1 and t_1 states are in a bisimilar relation to each other. It is however correct according to the definition. From the first state in both systems, it is only possible to take an a transition. The fact the there are two equivalent transitions does not matter as long as it is possible to simulate the further transitions in the same manner.

Figure 2.5: Example of bisimilar systems

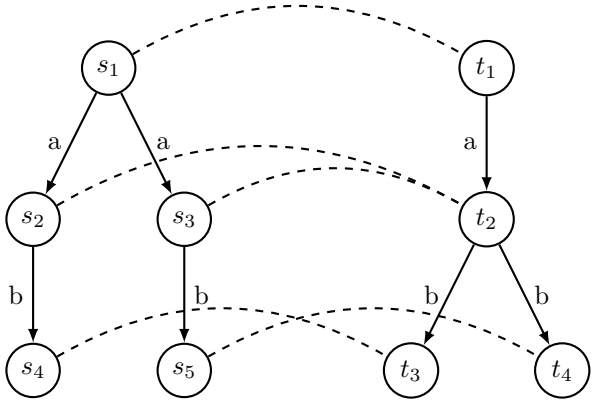
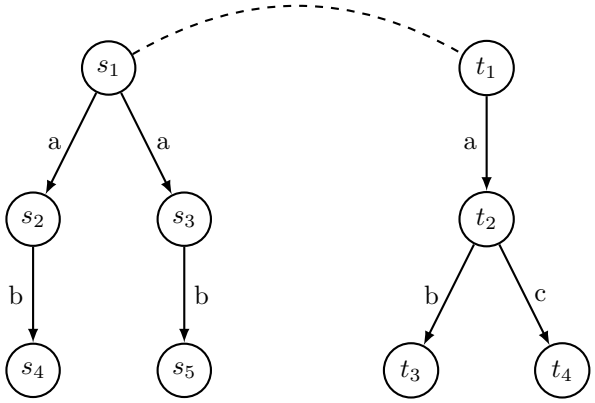


Figure 2.6 shows the slightly modified example where the systems are not bisimilar. It is visible that it is possible to take the b transition in state t_2 as it is in states s_2 and s_3. It is however not possible in s_2 and s_3 to take the c transition, which breaks the second requirement of the definition.

Figure 2.6: Example of systems that are not bisimilar



Chapter 3

Method

Testing has always been an important part of software development process. It increases the quality and safety of products and reduces time spent finding bugs in the codebase. In the recent years, manual testing has been increasingly accompanied or even replaced by unit and integration testing. Modern technologies such as formal verification cannot fully supplant the need for testing software and we still rely on it in the development pipeline. They can however increase assurance especially in safety critical areas.

Mathematical proofs are best in making sure the system will run as it is expected of it. They are however a rather abstract way of describing how systems work. Functional properties we specify are usually too general to give a full certainty that the deployed software will retain the qualities we expect of it. We can and do sometimes use formal verification to check detailed execution details, but the process is difficult and requires a lot of effort in implementing. We can automatically generate checks for some common properties such as memory safety[5]. Reasoning about the program execution however would require a tedious manual work of annotating code line by line and would be extremely expensive. In safety critical domains such as aeronautics this cost is justified, but in most cases an occasional crash is not a significant problem.

Model Driven Engineering tries to make the development more predictable and verifiable by focusing on the design of the application, verifying the assumptions are correct and generating the code that will put the promises to life. There is a visible disconnect however which is the object of this thesis.

The problem lies in the fact that by verifying the models we use to generate executable code we only look at the system as an abstract automaton without taking into account how it will behave on the actual machine. The generated

code can be tested but it is often very difficult or even impossible to express all the intricacies of the specification using a test case. What is missing in the MDE process, is the assurance that the generated code still behaves in the same way as the model from which it is translated. The problems may stem from bugs in the transformation stage or the difference between how a construct can be captured using formalisms in the model and the result language.

The goal of this work is to solve this issue by ensuring that the transformed code preserves the behaviour of the original model. The low hanging fruit in this case would be to restrict ourselves to translating the functional properties described for the model and use some of the model checking technologies to verify that they still hold in the source code. This however would give us an answer only in frame of specific properties we describe. Instead we have decided to verify that the semantics of the application will be exactly the same before and after the translation. This so-called *semantics preservation*[13] of the translation would ensure that any property that is provably true on the model will also be true in the result.

One of the main problems with the issue at hand is that of the previously mentioned difference between the transformed languages. Models are usually an abstraction of the reality. If they would describe every detail of the inner workings of the system they would not be needed as we could have just relied on code. As a result they do not contain all of the information we wish to have to create a reliable executable version of them. This semantic gap[29] is often problematic as it is almost impossible for a machine do deduce the abstract meaning of a specific concept in the model. In effect our solution had to be immune to this problem and bridge the difference between the languages.

3.1 Control Flow Graphs

To investigate how to solve the issue, one has to look at the end product of any software project. Modern day computers are machines that were designed to execute a series of instructions one by one[31]. Programs can be written in many programming languages using many paradigms such as object oriented or functional. In the end however, no matter how they are described, they end up being executed by the processor step by step. This is the most granular way of describing software and we decided to use it to solve the issue.

In a similar way, every line of a Java program contains one or multiple instructions that have to be executed and that change values stored in the memory of the computer. The instructions may branch or jump to different sections of

the program, we can thus treat them as atoms of the execution process. We can further connect the atoms in the same order as it would occur in a program to obtain a representation of a program that is useful for analysis. Such a structure is called a Control Flow Graph[2] and is used broadly in compiler development.

What is important about the CFG is the observation that given a specific state of memory, the path taken through the graph for that specific state will always be the same because the conditions that are to be satisfied will depend on it. This extends to the fact that if Control Flow Graphs of programs are identical and the states of their variables at the start are the same, they will represent the same behaviour and will be semantically equivalent.

3.2 The Tool

To verify whether the validity of functional properties of an SLCO model has been preserved in the translated Java source code, a number of actions has to be performed. To facilitate these steps, we have written an application `CFG-Tool` that will automate the process and require little user interaction. In effect, the checking process would become easy to use and can be incorporated into the development pipeline.

The tool will translate the given SLCO model or a Java file to a set of Control Flow Graphs that are representative of the system behavior. In case of an SLCO model, each statemachine will be translated to a separate graph. Similarly, in case of a Java source code, each subclass of a thread that was generated from a statemachine will be translated to its own CFG. The choice of generating and analyzing the statemachines in isolation comes from the way the interactions between them are described in the SLCO semantics. We further argue that this is correct in chapter 5.

The generated graphs will contain all the information about the state of variables at the start of the execution. They will also contain all the operations performed after the system has been initialized. Finally, the flow between those operations will be preserved by the graph. Having done that, by comparing respective graphs generated from SLCO and Java we will be able decide if the total behavior presented by both of them match.

If at all the points, the statements that are to be executed are the same and we can assume that no external change to the state might happen, the semantics of both systems are equivalent. What follows from this observation is that any property that is verified on one system will also hold on the transformed system. For transformations that do not change the semantics of the original model we

say that they are *semantics preserving*[13]. This releases the need to again verify the generated code for the functional properties checked on the original model.

The property that two systems have the same potential behaviour, clearly resembles the concept of bisimulation. If we treat the statements as actions we can solve the problem of behavioral equivalence by applying a bisimilarity checking algorithm. If we assume that states of both sides are the same at the beginning (all of the variables are undefined at the start of the program), we can then track that the changes to both states are exactly the same during the course of both SLCO and Java executions.

3.3 Semantic Gap

It becomes obvious that not all models can be directly expressed in terms of instructions and memory. The purpose of relying on MDE is based on abstracting information and due to that the detailed execution details are left undetermined. What it means is that this information is not present in the original model that is to be translated to the executable code.

To fill in the missing parts, we again have to turn to how the programs are being run on a real world machine. When translating the model into Java code, the developer of the translation step has to manually fill in all the missing parts to define how to execute the behaviour described by the model. Unfortunately there is no automatic method that can be used to deduce how those parts should look like. Given this fact it has to remain a manual process.

In our case, we have to bring two different languages to the same level of abstraction. Choosing the higher abstraction would result in ignoring details that can influence the semantics of the program. We choose the lowest abstraction of the two (imperative code in this case) and simplify it to reduce any information that is not essential to describe the behaviour.

To bring both model and its translation to the common form, we have decided to introduce an Intermediate Representation language that is based on the notion of changing the state of the program. The factor most descriptive on how the program behaves is the state of the variables and we describe it in terms of changes to this state. Furthermore we have to take into account that programs do not have to be linear and we model control flow using branching statements and loops.

3.4 Atomicity

The notion of atomicity is used in the domain of parallel systems to describe actions that represent a logical unit of behavior and cannot be interrupted or influenced by any other action in the system. Related to the concept is the notion of a race condition which describes a situation when two different executions will compete for the access to a variable resulting in an undefined behavior.

When developing the tool we ignore the notion of atomicity. We do not implement any special checks for the preservation of the atomicity property. Furthermore, we do not translate the atomicity guarantees described by the SLCO model to neither the IR nor CFG. SLCO language provides a composite statement type that allows for definition of a sequence of assignments that are not to be interrupted. Furthermore the assignment and expression statements themselves cannot be interrupted. In our translation we straightforwardly translate the composite statement in the same manner as we would an expression and sequence of assignments.

The reason for this omission is the work of Zhang in [33]. In the thesis she describes the development of a mechanism that ensures the preservation of atomicity with the translation. By the use of fine-grained ordered locking the translation of Composite statements is postulated to retain the atomic form described in the original model. This assumption is then proven using a specification developed in VeriFast[27] for the Java mechanism.

The Java translation that we use as a target of our check re-uses the same concept of achieving atomicity preservation. We use the results of [33] and assume no additional checks are needed as the original work proves the preservation for translations of all possible SLCO models.

We further argue that since we have thus far shown that the behavior is preserved between the SLCO model and the CFG, the introduction of a locking mechanism does not influence this representation. In the Java source code the locking mechanism is transparent and separated from the rest of the system using `Keeper` class. In chapter 5 we show that the `Keeper` class is not accessing any of the system variables nor changing the flow of the application, we then conclude that it is safe to exclude it from our analysis.

Chapter 4

Implementation

4.1 Platform

The project has been in written in the Haskell programming language[16]. Haskell is a purely functional programming language with strong static typing. Strong static typing requires variables to be annotated with precise types they will represent, with the compiler ensuring that the type will not change at any point. Furthermore all values in Haskell are immutable, which means that they are constant and cannot be re-assigned. This style of typing, contributes to what is called *referential transparency*[21], which means that any variable can be replaced by the corresponding value without changing the behaviour. We can safely reason about a value at any given moment without the danger of it changing in the meantime. A program written in Haskell is compiled as opposed to interpreted, which results in high performance, needed when dealing with complex models.

The language was chosen for a number of reasons. Primarily, it is well suited for working with parsing and languages. An important feature for parsing and analyzing programs is the pattern matching. Haskell makes heavy use of *sum types*, which permits to express a value as a list of different exclusive descriptions also called constructors. Pattern matching permits distinguishing the input type constructor when writing function. As a result we can model language constructs into a sum type and recursively perform specific operations based on the construct given.

Haskell also provides features that minimize the chance of writing erroneous code. The aforementioned strong static typing together with referential transparency provide a efficient way of reducing bugs in compile time.

Finally the vast ecosystem of the Haskell language, provides a number of libraries that handle complicated tasks, such as building parsers, handling graph structures and generating visualizations. Since the language has been in development for almost 30 years, it provides a stable environment to work in. By using the recent Glasgow Haskell Compiler 7.10.2, we ensure no compiler error occurs during the building of the project.

For bisimilarity checking we use mCRL2 toolset[7]. The toolset contains a command line program `ltscompare` that given two labelled transition systems will decide if they are strongly bisimilar. It also provides an option to give a counterexample in case no bisimilarity is established.

4.2 Input

The tool provides three commands to the user. Given a command `slco` and an SLCO model it will generate a visualization of a Control Flow Graph of all the statemachines in the given model. Similarly given a command `java` along with Java source code, it will generate a visualization of all the statemachine classes in the source. The third option available is the `compare` command, which given an SLCO model and a Java file will generate a CFG for every statemachine in the SLCO model along with the CFG for corresponding class in Java.

The SLCO models that are given to the verification tool are adhering to the SLCO 2.0[24] specification. As SLCO 2.0 does not yet generate any code related to the Channel structures we do not take into account such structures in the tool.

The Java code input is equivalent to the output of the code generated using the SLCO 2.0 framework. The main class is equivalent to SLCO model construct, while all the statemachines are represented in Java as a separate subclass of the `Thread` Java class. Each of those classes define a `run` method, which will serve as a basis for the separate Control Flow Graph.

4.3 Output

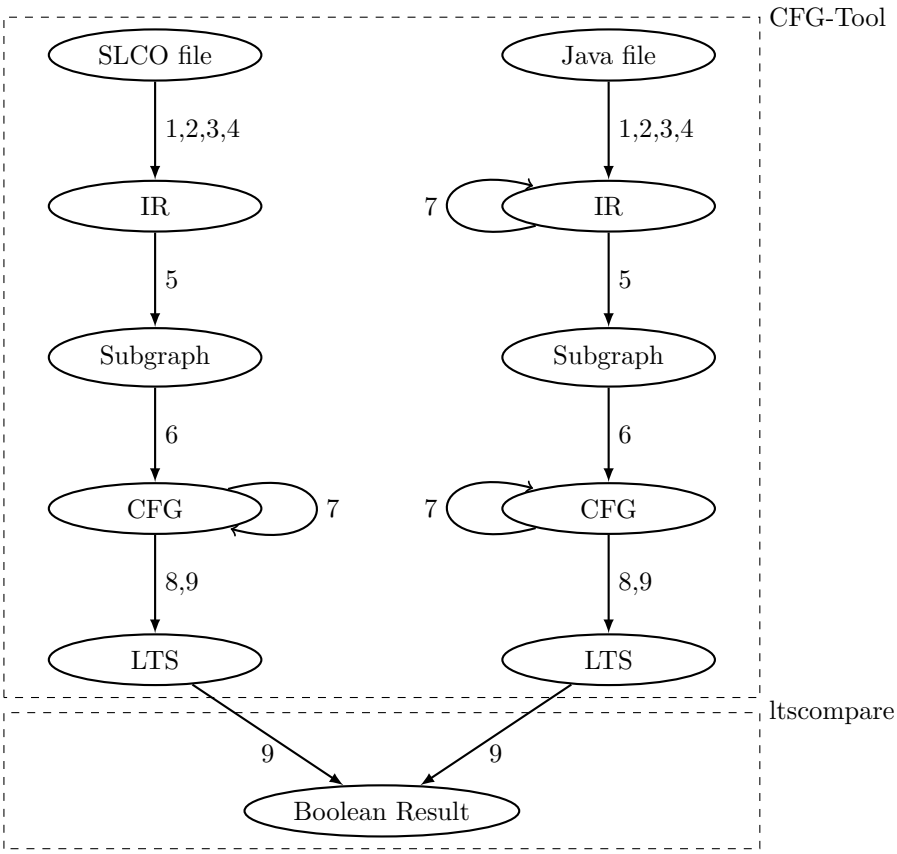
Given both the SLCO model and the Java source code that was generated from it, our goal is to provide a definitive yes or no answer whether the systems are equivalent in terms of semantics. As we are using a third party tool to check the bisimilarity, the output of the `CFG-Tool` program will be a set of labelled transition systems imitating a CFG structure. The difference between the original Control Flow Graph and the LTS is that all the data contained in

the nodes of the CFG will be moved to the edges of the graph as to imitate the actions in the transition system. This enables the compatibility with the `ltscompare` tool without losing any of the original information encoded in the graph. The final output will be formatted to the Aldebaran format accepted[11] by the tool.

4.4 Workflow

The checking procedure consists of 10 steps as shown on figure 4.1

Figure 4.1: Procedure of Verifying Semantics Preservation



1. Parsing - where both SLCO model and Java source are read and stored in memory.
2. Extracting - where parts of the SLCO model/Java code representing execution of a statemachine are extracted and separated. From this point,

all operations are performed for each statemachine.

3. Translating to Intermediate Language (IR) - where the memory representations are translated to the common language to bring them to the same abstraction level.
4. Building the Abstract Syntax Tree (AST) - where the IR elements are joined to form a logical structure.
5. Traversing the AST and Building The Subgraphs - where the AST structures are being transformed into subgraphs representing the CFG basic blocks.
6. Generating the Control Flow Graph - where all of the subgraphs are merged into an unified structure.
7. Running the optimizations - where unnecessary nodes are being removed and the graph is simplified.
8. Rewriting to Labelled Transition System - where all nodes are being moved to the newly created edges.
9. Generating Aldebaran format file - where the LTS is being formatted for input into mCRL2.
10. Checking bisimilarity - where `ltscompare` will check the corresponding CFGs of Java and SLCO statemachines for bisimilarity.

4.5 Running Example

To better visualize the process of translating SLCO model and Java code to the CFG we have prepared a simple SLCO model. The model contains only one object and one statemachine to reduce the complexity. The statemachine contains only two states and three transitions out of, which two are starting at the same state, which should visualize how the nondeterministic transition is modelled in the CFG. Furthermore we restrict ourselves to only two types of statements - expression and assignment. SLCO statemachine contains initialized variables to present how the translation process handles the starting state of the system. The example SLCO model is presented on listing 4.1. The Java source code has been generated using the SLCO 2.0 translation framework[24]. The statemachine constructor and main function are presented on listing 4.2.

4.6 Parsing the Code

Before any kind of analysis may happen, the input SLCO model and Java source code have to be loaded and understood by our application.

Listing 4.1: SLCO Running Example

```
model Example {
        classes
        A {
          state machines
                SM {
                  variables Boolean x:=False Integer i:=2
                        initial S0 states S1
                        transitions
                                from S0 to S1 {
                                        x; i := i+1;
                                }
                                from S0 to S0 {
          i := 0;
        }
        from S1 to S0 {} }}
          objects a: A()
}
```

4.6.1 SLCO

The SLCO parsing is performed using a technique called parser combinators[14]. Parser combinators are functions that accept other parsers as an input and return parsers as output. This enables extensive modularity and writing parsers that closely resemble the original Backus–Naur form grammar. To reduce time to develop the solution the `Megaparsec`[15] library is used. The library defines basic functions that will parse characters, numbers and provide error handling in case the parsing has failed.

The parsing logic is achieved by defining parsers for basic structures of the language, such as keywords, symbols and operators. One way of defining those structures (called primitives) is to use predefined character parsers. By using `lexeme` or `symbol` we can define the keywords and operators to form a basic lexer. The primitives are then used together with combinator operators to form more complicated constructs such as class or expression. The operator used the most when dealing with languages would be the alternative operator `<|>` that chooses the first parser that successfully matches the input. One other example of operator is `option`, which in case the following parser matches will return the result of such parsing, otherwise it will fallback to the given default value. More advanced examples of operators include `sepBy`, which will parse instances of a different combinator separated by a given symbol. Similarly `between` will apply a parser for text that is surrounded by two symbols, this proves useful in case of parsing parentheses or array indices.

Listing 4.2: Java Running Example

```
java_SMThread (Example.java_Keeper java_k) {
        java_randomGenerator = new Random();
        java_currentState = Example.java_State.S0;
  java_kp = java_k;
  java_lockIDs = new int[0];
        x = False;
        i = 2;
}

public void exec() {
        int java_choice;
        while(true) {
                switch(java_currentState) {
                        case S0:
                                java_choice = java_randomGenerator.nextInt(2);
                                switch(java_choice) {
                                        case 0:
                                                Arrays.sort(java_lockIDs,0,0);
                                                java_kp.lock(java_lockIDs, 0);
                                                if (!(x)) { java_kp.unlock(java_lockIDs, 0);
                                                ↪  break; }
                                                java_kp.unlock(java_lockIDs, 0);
                                                Arrays.sort(java_lockIDs,0,0);
                                                java_kp.lock(java_lockIDs, 0);
                                                i = i + 1;
                                                java_kp.unlock(java_lockIDs, 0);
                                                java_currentState = Example.java_State.S1;
                                                break;
                                        case 1:
                                                Arrays.sort(java_lockIDs,0,0);
                                                java_kp.lock(java_lockIDs, 0);
                                                i = 0;
                                                java_kp.unlock(java_lockIDs, 0);
                                                java_currentState = Example.java_State.S0;
                                                break;
                                }
                                break;
                        case S1:
                                java_currentState = Example.java_State.S0;
                                break;
                        default:
                                return;
                }
        }
}
```

Lexer

We start constructing the parser with building a basic lexer for the language. First thing we take into account is that in SLCO whitespace characters have

no meaning apart from separating tokens. Additionally, we do not take into account the number of whitespaces between those tokens. We define a space consumer `sc` parser to handle those whitespaces as shown on listing 4.3. The whitespace parser is created using the `L.space` function that takes three other parsers - one for space characters, one for line comments and one for block comments.

Listing 4.3: Space Consumer Code

```
sc :: Parser ()
sc = L.space space1 lineCmnt blockCmnt
  where
    lineCmnt  = L.skipLineComment "//"
    blockCmnt = L.skipBlockComment "/*" "*/"
```

We then can use the space consumer to define parsers for lexemes, symbols (which match a specific exact string), and integers (listing 4.4).

Listing 4.4: Space Consumer Code

```
lexeme :: Parser a -> Parser a
lexeme = L.lexeme sc

symbol :: String -> Parser String
symbol = L.symbol sc

integer :: Parser Integer
integer = lexeme L.decimal
```

Finally we write a parser for identifiers. The important thing is to make it impossible for an identifier to be any of the reserved words for the language (listing 4.5).

The parser for identifier (listing 4.6) is then any word or combination of word and a number that is not part of the reserved words list.

SLCO Grammar

SLCO 2.0 provides a language for textual representation of a model . In the framework SLCO grammar is defined using `textX` format (appendix A).

Listing 4.5: SLCO Reserved Words

```
rws :: [String] -- list of reserved words
rws = [
  "actions", "model", "classes", "ports", "state machines", "variables",
  ↪  "initial", "state", "transitions", "from", "to", "send", "receive",
  ↪  "objects", "channels", "Boolean", "Integer", "Byte", "async",
  ↪  "sync", "lossless", "lossy", "between", "and", ":=", "after", "ms",
  ↪  "not", "ms", "-", "+", "or", "xor", "and", "==", "<>", "<=", ">=",
  ↪  "<", ">", "mod", "*", "/", "**", "{", "}", "(", ")", ":", "true",
  ↪  "false"]
```

Listing 4.6: SLCO Identifier Parser

```
identifier :: Parser String
identifier = (lexeme . try) (p >>= check)
  where
    p       = (:) <$> letterChar <*> many (alphaNumChar <|> char '_')
    check x = if x `elem` rws
                then fail $ "keyword " ++ show x ++ " cannot be an
                ↪  identifier"
                else return x
```

In the `CFG-Tool`, the grammar structures are being parsed in a monadic fashion. What it means is that we perform the parsing in a specific context with the ability to bind intermediate results of the parsing to specific variables. Listing 4.7 shows how the main model is being parsed using this technique. First the parser will try to match a keyword `actions` followed by a list of action identifiers. If successful it will store the resulting identifiers in a variable `act`, otherwise the variable will evaluate to an empty list. The parser will then expect the `model` keyword and store its identifier in the `name` variable. The parser will then expect a list of classes and objects parsed by external parsers `slcoClass` and `slcoObject`. Finally the function will return the representation of the model containing all the parsed information.

A similar method is used in SLCO class and object parsers.

Listing 4.7: SLCO Model Parser Function

```
slcoModel = do
  act <- option [] (symbol "actions" *> many (Action <$> identifier)
  symbol "model"
  name <- identifier
  symbol "{"
  cls <- symbol "classes" *> many slcoClass
  let classes = Map.fromList $ map (\x-> (className x, x)) cls
  obj <- symbol "objects" *> many slcoObject
  symbol "}"
  return (Model act name classes obj [])
```

Expressions

The common issues with parsing expressions in any language are those of left recursivity, precedence and operator association. Thankfully `Megaparsec` provides a useful function to define expression parsers without having to take care of those problems. Listing 4.8 presents how the `makeExprParser` can be used along with a table to define an expression parser sorted by precedence and with associativity defined for each expression.

4.6.2 Java

Parsing Java is achieved using `language-java`[18] library. The library provides a `parse` function, which given Java source will parse it to a memory representation, which then can be traversed or translated.

4.7 Bridging the Semantic Gap

One of the problems mentioned in the beginning of this chapter is that of the possible difference of abstractions between the source and target models. To solve this issue we need a way to bring the semantics of both languages to a common denominator. This is achieved by defining an Intermediate Representation (IR) language that will preserve the constructs of both of the source and target.

Listing 4.8: SLCO Expression Parsing

```
exprTable = [
  [prefix "-" NegExpr, prefix "not" NegExpr],
  [binary "*" MulExpr, binary "/" DivExpr, binary "%" ModExpr],
  [binary "+" SumExpr, binary "-" DiffExpr],
  [binary "!=" NeqExpr, binary "=" EqExpr,
   binary "<>" NeqExpr,
   binary "<=" LeqExpr, binary ">=" GeqExpr,
   binary "<" LeExpr, binary ">" GeExpr],
  [binary "or" OrExpr, binary "||" OrExpr,
   binary "and" AndExpr, binary "&&" AndExpr,
   binary "xor" XorExpr]]

term = GroupExpr <$> parens expr
   <|> RefExpr <$> slcoVarRef
   <|> LitIntExpr <$> integer
   <|> LitBoolExpr True <$ symbol "True"
   <|> LitBoolExpr False <$ symbol "False"

expr = makeExprParser term exprTable
```

4.7.1 Intermediate Representation

SLCO representation focuses mainly on the structure and the order of actions. In contrast, Java representation has more granularity in terms of how those actions will be performed by the computer. Taking this into account the IR language has to be semantically closer to the Java code as it is possible to express all SLCO constructs using Java code but not the other way around. We make a small exception to this statement by defining an explicit nondeterministic choice instruction to directly define nondeterministic behaviour in SLCO but it needs a more complicated mechanism in Java. We define the Intermediate Representation program as a sequence of instructions shown on listing 4.9.

The instructions represent the following:

- `Conditional` - a conditional statement that will execute a number of instructions if the condition is satisfied.
- `Branch` - a statement that will branch to multiple other statements depending on the result of the condition. This mirrors the `if else` construct of Java.

Listing 4.9: IR instructions

```
data Ins = Conditional Expr Block
         | Branch Expr [(Expr, Block)]
         | Switch Expr [(Expr, Block)]
         | Nondeterm [Block]
         | Loop Expr Block
         | Assign VariableRef Expr
         | MethodInv MethodCall
         | Exp Expr
         | BlockIns Block
         | Break
```

- `Nondeterm` - a statement which will branch to multiple other statements. The chosen path is not depending on any rule and an arbitrary branch can be chosen instead.
- `Loop` - a statement that will repeat for as long as the condition is satisfied.
- `Break` - a statement that will break the current computation and link directly to the next instruction following the surrounding Switch or Loop.
- `Switch` - an extension of the Branch statement. The difference is that Break statement will only break the Switch and not the Branch.
- `Assign` - an assignment statement.
- `MethodInv` - a named method invocation.
- `Exp` - an expression statement.
- `BlockIns` - a sequence of statements.

4.7.2 Translating SLCO to IR

The SLCO translation to the IR is heavily influenced by the way the Java code is generated. We want the end representations of both SLCO and Java to be as close as possible. Since SLCO is of higher abstraction than Java, we specify all the missing details during the translation to IR. The code that is generated by the Java generator includes a separate `Thread` object for each SLCO statemachine. In IR translation, we thus take each statemachine and convert it to a sequence of instructions.

To reduce the size of the control flow graph, as well as to simplify the translation process, we use the results of [33] that tell us that the atomicity properties will be preserved between the SLCO model and the target Java source code.

This releases the need to include any information with regards to atomicity in our translation.

The first thing we need to take into account is the state of the statemachine at the start of execution. We handle this by defining a `currentState` variable to track the current state of the statemachine. We then assign the starting state to the variable and translate all of the variable initializations to IR `Assign` instructions. We then need to handle the fact that a statemachine will continuously run when executed, which we model using an infinite `Loop` instruction with `True` as the expression that will be checked each time we complete a transition. Finally we construct a `Switch` instruction that will contain a case for all the possible starting states of all transitions. Depending on the number of the outgoing transitions of the selected state, we construct in each case

- If there is only a single transition starting at the selected state we proceed to translate the statements in sequence.
- If there are multiple transitions coming out of the same state it means there is a nondeterministic choice of transition. We construct a `Nondeterm` instruction that will represent this choice. In each case of the `Nondeterm` instruction we translate the statements of the respective transition.

We finalize the translation of each transition by assigning the final state of the transition to the `currentState`.

As Java code generation for Channels is not currently implemented in the SLCO framework, we skip the translation of Channels and and channel related SLCO statements (send signal, receive signal and delay). This leaves us with 3 types of SLCO statements that need to be translated:

- Assignment has its own direct counterpart in the IR, it is translated to the `Assign` instruction.
- Expression statement will have a different translation depending on its placement in the transition instructions. If it is the first statement in the transition, we construct a `Conditional` instruction that will check if the expression evaluates to false and return to the main loop if it is the case. If the expression is not the first statement in the transition, it translates to a `Loop` statement that checks on the negation of the expression continuously. As soon as the expression evaluates to true we can continue with the execution.
- Composite statement defines a sequence of assignments with an optional guard that are supposed to happen as an atomic step. We have decided we do not need to check for atomicity preservation. We can thus translate the statement to `BlockIns` of translated assignment statements with optional

expression condition in the beginning.

Let us revisit the running example and see what the generated IR code looks like for the simple SLCO model (listing 4.10). The translated IR starts with a assignment of the current state. It then initializes the statemachine variables. The main loop contains a `Switch` on the `currentState` value. Since at the state `S0` occurs a nondeterministic choice, we can observe the `Nondeterm` instruction, which contains cases for two possible transitions from this state. The first transition is guarded by the condition, which gets translated to a `Conditional` expression, which will `Break` and return to the main loop in case the guarding condition does not evaluate to true. Each of the transitions then translate the assignments associated with them to the respective `Assign` instructions.

Listing 4.10: IR translation of the running SLCO example

```
[Assign (LitStringExpr "currentState") (LitStringExpr "S0"),
 Assign (LitStringExpr "x") (LitBoolExpr False),
 Assign (LitStringExpr "i") (LitIntExpr 2),
 Loop (LitBoolExpr True) [
  Switch (RefExpr "currentState") [
    (RefExpr "S0", [Nondeterm [
      [Conditional (NegExpr (RefExpr "x")) [Break], Assign (LitStringExpr "i") (SumExpr
      ↪  (RefExpr "i") (LitIntExpr 1)),Assign (LitStringExpr "currentState") (LitStringExpr
      ↪  "S1")],
      [Assign (LitStringExpr "i") (LitIntExpr 0),Assign (LitStringExpr "currentState")
      ↪  (LitStringExpr "S0")]]]),
    (RefExpr "S1",[Assign (LitStringExpr "currentState") (LitStringExpr "S0"),Break])
  ]]]
```

4.7.3 Translating Java to IR

Java translation begins with finding the function that describes the respective statemachine execution. Every `Thread` subclass defines a `run` function, which describes what should be done when the thread is started. But this would exclude the initial state of the variables belonging to the statemachine. The translation should thus start with translating the class constructor. The constructor contains all the variable initializations for the starting state, which we translate into IR. We then proceed to translate all the statements in the `run` method of the class one by one. The IR language instructions are in most cases simplifications of Java language constructs. For instance both `for` and `while` loops will translate to just `Loop` instruction. Similarly `switch` and `if then else` would translate to the `Branch` condition with notable exception of

`if` without an `else` clause, which will translate to `Conditional`.

An important distinction between SLCO and Java is that the latter does provide a possibility to define and call functions. Since function calls can contain some additional changes to the state that we have to take into account, we cannot just leave them in their original form. Since the whole of the program was parsed and stored in memory, we have the ability to find and expand the function call into series of statements from its definition. Translating function calls posed two important problems that we needed to solve. First, it is important to realize that not all of the used functions will be defined in the parsed file. Most Java applications use at least some of the classes and functions defined in the standard library or other libraries. Since we cannot easily obtain access to the source code of those functions, if we cannot find a definition we leave the method call as is. Another problem is that of method calls on class instances. Suppose we create an object of a certain class and call a method on it. At the call location, all the information with regards to the original class of the object is missing. We need to therefore keep track of types of all the objects that are declared in the source code. We solve the problem by storing a dictionary of all the variables defined in code together with their declared types. This enables us to query the class on which the method is called.

The opposite distinction is also true as Java has no way of directly defining the nondeterministic choice of a branch. The SLCO generator deals with the problem by assigning each of the possibilities a number and then generating a random value to decide, which of them to choose. We leave the generation as is during the translation stage and revisit the issue in the optimization phase.

Listing 4.11 shows how the running example Java code will get translated into IR. It is immediately visible that the Java version is much more complex than the SLCO version. This complexity is due to the fact that the grammar of Java is much more extensive than that of SLCO and the translation procedure has to accommodate for those differences. It is however visible that both translations are similar in form as they both contain the preamble where all values of the variables at the start of the system are defined, the main loop which will continuously execute the system and the main switch that will choose the transitions depending on the current state. Two major differences become visible upon inspection. First is the inclusion of method calls that are associated with the atomicity mechanism in the translated code, second is the missing nondeterministic choice instruction. Java language does not include a construct for the nondeterministic choice and instead the translation relies on the condition depending on the randomly generated variable. We will address those issues in

the next steps of the translation process.

Listing 4.11: IR translation of the running Java example

```
[Assign (LitStringExpr "java_currentState") (RefExpr "S0"),
 Assign (LitStringExpr "java_kp") (RefExpr "java_k"),
 Assign (LitStringExpr "x") (RefExpr "False"),
 Assign (LitStringExpr "i") (LitIntExpr 2),
 Loop (LitBoolExpr True) [BlockIns
  [Switch (RefExpr "java_currentState") [
    (RefExpr "S0",[
      Assign (LitStringExpr "java_choice") (FunctionResult (MethodCall
      ↪  "java_randomGenerator.nextInt" ["2"])),
      Switch (RefExpr "java_choice") [
        (LitIntExpr 0,
          [BlockIns [MethodInv (ClassMethodCall "Arrays" "sort" ["java_lockIDs","0","0"])],
          ↪  BlockIns [MethodInv (ClassMethodCall "java_kp" "lock" ["java_lockIDs","0"])],
          ↪  Conditional (NegExpr (RefExpr "x")) [BlockIns [BlockIns [MethodInv
          ↪  (ClassMethodCall "java_kp" "unlock" ["java_lockIDs","0"])],Break]],BlockIns
          ↪  [MethodInv (ClassMethodCall "java_kp" "unlock" ["java_lockIDs","0"])],BlockIns
          ↪  [MethodInv (ClassMethodCall "Arrays" "sort"
          ↪  ["java_lockIDs","0","0"])],BlockIns [MethodInv (ClassMethodCall "java_kp"
          ↪  "lock" ["java_lockIDs","0"])],Assign (LitStringExpr "i") (SumExpr (RefExpr
          ↪  "i") (LitIntExpr 1)),BlockIns [MethodInv (ClassMethodCall "java_kp" "unlock"
          ↪  ["java_lockIDs","0"])],Assign (LitStringExpr "java_currentState") (RefExpr
          ↪  "S1"),Break]),
        (LitIntExpr 1,
          [BlockIns [MethodInv (ClassMethodCall "Arrays" "sort"
          ↪  ["java_lockIDs","0","0"])],BlockIns [MethodInv (ClassMethodCall "java_kp"
          ↪  "lock" ["java_lockIDs","0"])],Assign (LitStringExpr "i") (LitIntExpr
          ↪  0),BlockIns [MethodInv (ClassMethodCall "java_kp" "unlock"
          ↪  ["java_lockIDs","0"])],Assign (LitStringExpr "java_currentState") (RefExpr
          ↪  "S0"),Break])],Break]),
    (RefExpr "S1",
      [Assign (LitStringExpr "java_currentState") (RefExpr "S0"),Break]),
    (Default,[Effect "RETURN"])
    ]]
  ]
]
```

4.8 Generating the Control Flow Graph

The process of generating the CFG from both SLCO and Java Intermediate Representation is split into two parts. First the IR instructions are being converted one by one into data structures similar to the aforementioned basic blocks. They can have however multiple exit points. We call these structures subgraphs as they are used to build a bigger Control Flow Graph. We then proceed to merge the subgraphs sequentially to form one resulting CFG.

4.8.1 Subgraph

We define `Subgraph` (listing 4.12) as a $SG = (starts, ends, nodes, transitions)$ where:

- *start* - is the id of the first node in the subgraph.
- *ends* - is a list of ids of the last nodes in the subgraph.
- *nodes* - is a list of nodes contained in this subgraph.
- *transition* - is a list of transitions in the subgraph.

Every node is defined as $N = (id, contents)$, where *id* is the integer identifier of the node that is unique during the whole translation process. The value of *contents* is one of the following:

- `Condition` - is a node that will direct the flow of the program to one of the outgoing edges depending on the result of an expression.
- `Nondeterm` - is a node that will represent a nondeterministic choice of one of the outgoing edges.
- `Assignment` - is a node that will change the value of the given variable.
- `Effect` - is a node that represents any other instruction, which does not directly change variable values or the program flow i.e. Java function call or user defined action.

Every transition is defined as $T = (source, edge, target)$, where *source* and *target* are the identifiers of nodes connected by the transition and *edge* is one of the following:

- `Fallthrough` - is an edge that represents an unconditional flow that happens because of the order in which instructions are executed.
- `ExprEdge` - is an edge that represents a certain expression. These edges are coming out of the `Condition` nodes and represent the different values the condition can evaluate to.
- `Choice` - is an edge coming out of the `Nondeterm` node representing one of the possible nondeterministic choices that may be taken.

4.8.2 Translating Intermediate Representation

Given a list of instructions obtained by translating either SLCO or Java file to Intermediate Language, we transform them one by one into a Subgraph structures.

Conditional

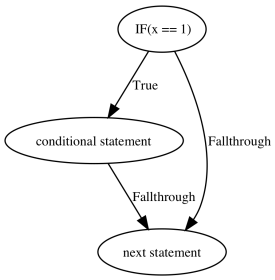
Conditional instruction gets transformed to a condition node and the statement that will be executed if the condition evaluates to true. If the condition is not

Listing 4.12: Subgraph Definition

```
type Id = Int
type Transition = (Id, Edge, Id)
type Node = (Id, NodeContents)

data Edge = Fallthrough
          | ExprEdge IR.Expr
          | Choice

data NodeContents = Condition IR.Expr
                  | Assignment IR.Expr IR.Expr
                  | Effect IR.Expr
                  | Nondeterm

data Subgraph = Subgraph {
  start:: Id,
  ends:: [Id],
  nodes:: [Node],
  transitions:: [Transition]
}
```

satisfied the execution will fall through to the next executable statement.

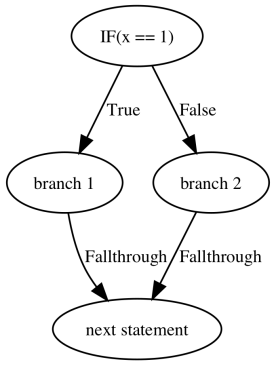
```
[IR.Conditional
  (IR.EqExpr (IR.LitStringExpr "x")
  ↪  (IR.LitIntExpr 1))
  [(IR.Effect "conditional statement")],
IR.Effect "next statement"]
```



Branch

Branch instruction is translated into a condition node and a list of statements that get executed in case the branch expression evaluates to true. All of the branch statements will fall through to the next executable statement.

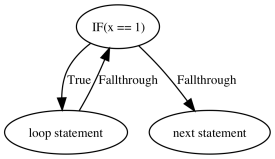
```
[IR.Branch
  (IR.EqExpr (IR.LitStringExpr "x") (IR.LitIntExpr
  ↪  1))
  [(IR.LitBoolExpr True, [IR.Effect "branch 1"]),
   (IR.LitBoolExpr False, [IR.Effect "branch 2"])],
IR.Effect "next statement"]
```



Loop

Loop instructions get translated to a conditional instruction representing the loop invariant and a list of conditional statements that will be executed if the condition is satisfied. The conditional statements will fall through back to the condition, which is also directly connected to the next statement.

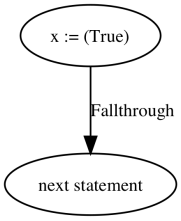
```
[(IR.Loop
    (IR.EqExpr (IR.LitStringExpr "x")
    ↪  (IR.LitIntExpr 1))
    [IR.Effect "loop statement"])
, IR.Effect "next statement"]
```



Assign, Exp, MethodInv

Assignment, expression and method invocation is translated to a single node representing the instruction. The node is being directly connected to the next statement.

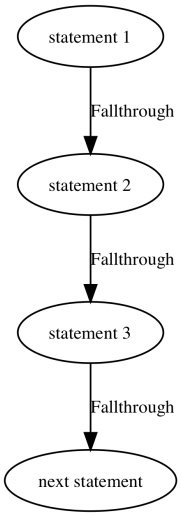
```
[IR.Assign "x" (IR.LitBoolExpr True),
IR.Effect "next statement"]
```



BlockIns

Block instructions can be thought of as a sequence of other IR instructions executed one after another. They are most often used in other IR instructions such as branches and switches. They will be recursively transformed into their own respective Subgraphs and merged together with fallthrough edges.

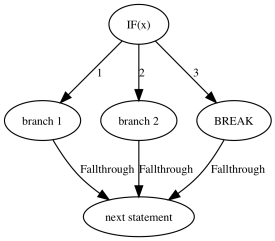
```
[IR.BlockIns [IR.Effect "statement 1", IR.Effect
↪  "statement 2", IR.Effect "statement 3"],
IR.Effect "next statement"]
```



Break and Switch

The subgraph generated from switch is almost completely identical to the one obtained from the branch instruction. The only difference lies in the fact that the switch can be interrupted using the break statement. When used, the break instruction will ignore any structure surrounding it and connect directly to the statement following the closest switch statement that surrounds it.

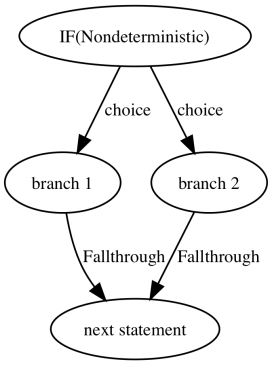
```
[IR.Switch (IR.LitStringExpr "x")
  [(IR.LitIntExpr 1, [IR.Effect "branch 1"]),
   (IR.LitIntExpr 2, [IR.Effect "branch 2"]),
   (IR.LitIntExpr 3, [IR.Break])
  ],
IR.Effect "next statement"]
```



Nondeterm

The nondeterministic instruction is transformed to a branching node with multiple edges coming out of it. Because an arbitrary edge can be chosen at any given time, we label all the edges with the same condition.

```
[IR.Nondeterm
  [
   ([IR.Effect "branch 1"]),
   ([IR.Effect "branch 2"])
  ]
IR.Effect "next statement"]
```



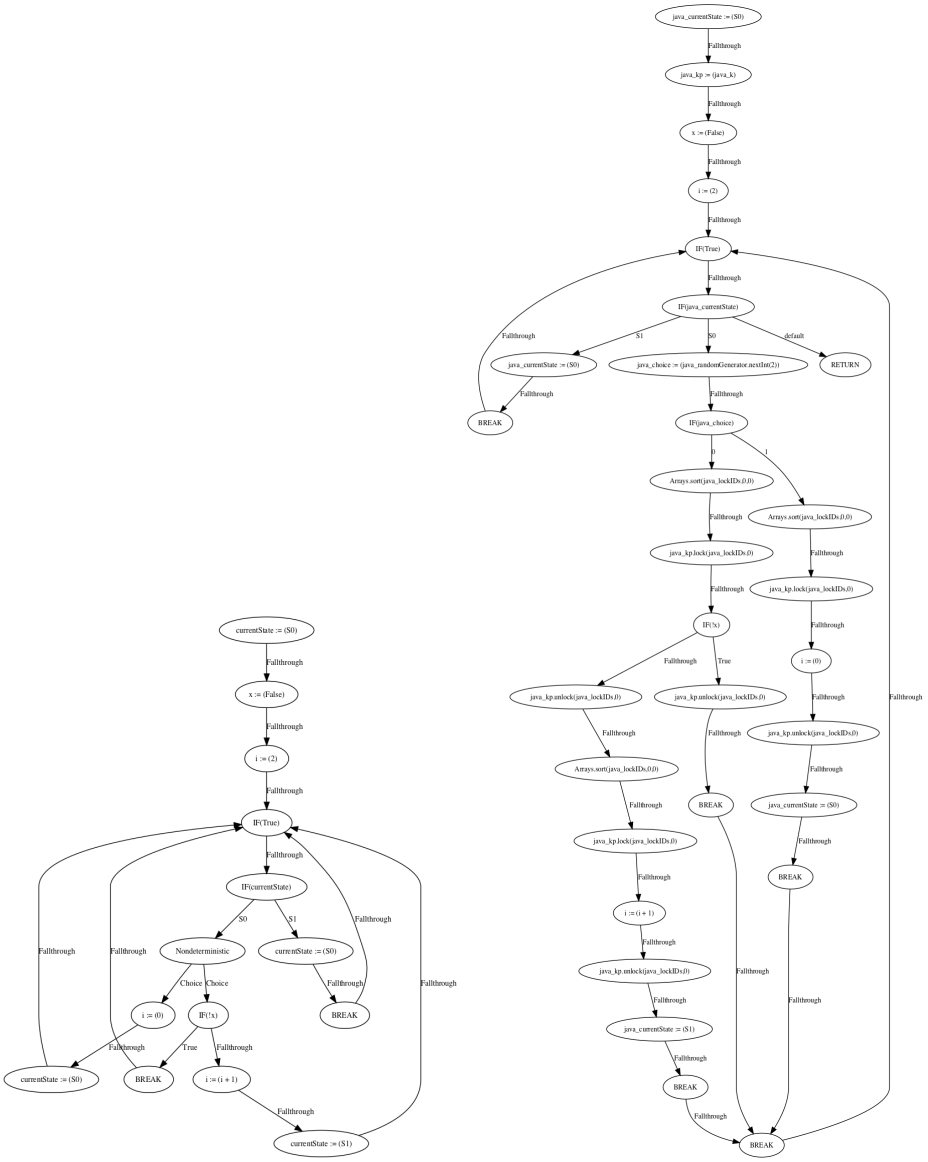
4.8.3 Merging Subgraphs

After translating Intermediate Representation instructions one by one, we obtain a list of Subgraphs that we need to connect together to form a proper Control Flow Graph. Since we store identifiers of the first and last nodes of each Subgraph, we create a new Subgraph that contains all of the nodes and transitions of the connected Subgraphs, set the beginning identifier to the beginning of the first Subgraph and all the end identifiers to the ones from the second Subgraph. Finally we generate a fall through transition between every ending node of the first Subgraph and the beginning node of the second.

The last step is to convert the obtained Subgraph into dedicated graph data structure imported from the Functional Graph Library[10]. The library defines functions that will enable querying and modifying the graph, which is useful when optimizing the graph structure as well as converting it to a Labelled Transition System.

To visualize the process of translating IR to subgraphs and merging them into an unified graph, on figure 4.2 we present the CFG generated from the SLCO statemachine represented by the IR on listing 4.10 (left) and the respective CFG generated from the Java IR translation from listing 4.11 (right). As it can be seen from the figures, the CFGs retain the same differences as it was the case with the IR. Compared to the IR instructions however it is now visible how the flow of the program will proceed. The statements for each transition are now connected with edges that signify the order of execution. Furthermore cases responsible for different transitions will return to the main loop after the completion of the statements. The information encoded into the nodes are also simplified - no notion of `Switch` or `Loop` is present because the flow details are now localized to the nodes and edges using conditionals instead of encoding them in the recursive instruction types.

Figure 4.2: CFGs of the Running Example



4.9 Optimizing the Result

To reduce the complexity of the check and make sure we remove some of the last language specific details encoded in the graphs, we perform a number of optimization passes.

4.9.1 Removing Dead Code

The first optimization pass is the removal of the code that will never be reached (also called dead code). In the comparison of the graphs we are only interested in the behavior that actually occurs. Very often the dead code is caused by the language semantics and its inability to precisely establish what will happen next. In this case, Java may perform static analysis on the code when generating virtual machine instructions, but we are performing operations on the source code only. As such we need to analyze which branches will never be reached as some conditions will never be satisfied.

The removal of dead code is a well researched domain. In [2] the authors describe a method of control flow analysis that models the program as a graph and propagates the information such as variable assignments to the succeeding nodes. In our case however we are not interested in a complex analysis but rather a rudimentary method for removing branches that are impossible to be reached at any point.

To achieve this goal we perform two passes on the parsed Java program. In the first pass, every time we encounter a variable assignment we add the assigned value to the set of possible values for the variable. In the second pass we use the sets to check whether the branching statement enabling condition evaluates to an element in the respective set. In case it doesn't, we remove the condition altogether and proceed until we reach the end of the application.

To demonstrate the difference, figure 4.3 presents the running example Java CFG before (left) and after (right) applying dead code removal. It can be seen on the improved graph that the `default` branch in the main switch is removed.

4.9.2 Java Graph Refactorings

Some minor alterations to the Java generated graph have to be performed to bring the result closer to the SLCO version. First, SLCO Java code generator will prefix all the variables with `java_` string, we thus remove the prefix in all occurrences in code. Additionally the Java code will define a `Keeper` class that is used to handle the atomicity in the system. The class is used to obtain and release the locks needed to serialize parallel code. As argued in section 2.2.5 we have no need to prove the atomicity preservation hence the class and all of its uses can be removed along with all the references to `lockIds` that are used for serializability.

Next we take care of the nondeterminism in the Java graph. Since SLCO language is of higher abstraction than Java, the nondeterministic transition se-

lection had to be implemented in an imperative manner. Any time the set of enabled transitions is greater than one, the generated Java code will generate a random integer smaller than the possible number of choices. Depending on the value of this random variable the execution will then choose a respective transition and take it. This is however not mirrored in the SLCO generated CFG, hence the random variable generation has to be removed. Furthermore we translate the switch statement depending on the random variable to a nondeterministic choice as it is done on the SLCO graph.

The comparison between the graph before (left) and after (right) the optimization is shown on figure 4.4. The first difference that becomes apparent when examining the graphs is the replacement of the `currentState` conditional node by a nondeterministic choice. Furthermore the random variable is no longer present in the graph. Finally - the `java_` prefixes are missing, as is the use of the `Keeper` mechanism responsible for enforcing atomicity.

4.9.3 Removing Breaks

During the translation to the IR stage we have introduced a break instruction that will exit the switch case prematurely. The instruction was needed for the merging of the Subgraphs stage but it is not needed any more after the Control Flow Graph is constructed. We therefore remove it in the final graph and rewrite all the incoming transitions to target the successor of the break node.

We present two graphs before and after the effect of the `Break` removal on figure 4.5.

4.10 Checking Bisimulation

The last step of the method is to check the bisimulation between Java CFG and SLCO CFG. We accomplish this using the `ltscompare` tool present in the `mCRL2`[7] toolset. The tool can be used to check more than just strong bisimulation as it provides algorithm for checking weak bisimulation, trace equivalence and other variations. To be able to run the tool on the Control Flow Graphs however, they need to satisfy two conditions. First, they need to be structured as a Labelled Transition system and second they have to be formatted in the Aldebaran file format.

The task of transforming a graph to a Labelled Transition System is accomplished by creating a new transition for each of the nodes in the CFG. Say that the node N is defined as $N = (i, c, s)$ where i is the integer identifier of the

node in the graph, s is a list of the transitions coming out of the node and c is the contents of the node. For each node we create a transition T in the form of $T = (i, x + i, n)$ where x is any variable that is higher than the number of nodes in the graph. Then for each of the out transitions in the form $O = (a, b, l)$ where a is the source node identifier, b is the target node identifier and l is the edge joining the two states, we rewrite them to the form $(x + a, b, l)$. The final LTS is shown on figure 4.6

Finally to obtain the LTS in Aldebaran format we first rewrite all the node identifiers to be unique and sequential without gaps. We then print them out as specified by the Aldebaran grammar.

For each of the statemachine, the properly formatted LTS of an SLCO statemachine and the corresponding Java code are passed to the `ltscompare` tool using `-e bisim` flag to set the equivalence mode to strong bisimulation. The system will verify that all the statemachines and code match. The final answer is then presented in a message on the command line.

Figure 4.3: Dead Code Removal

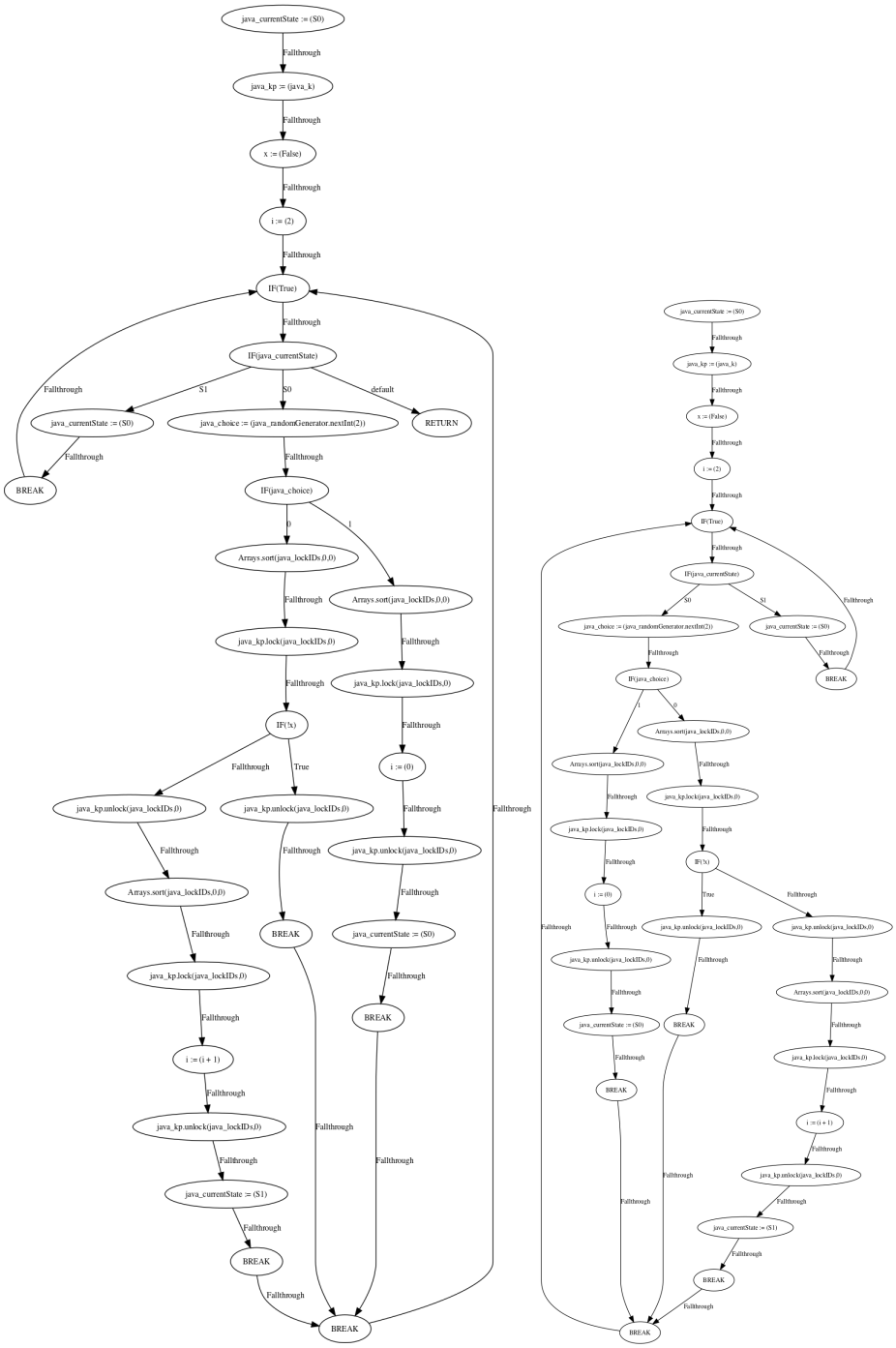


Figure 4.4: Java Refactorings

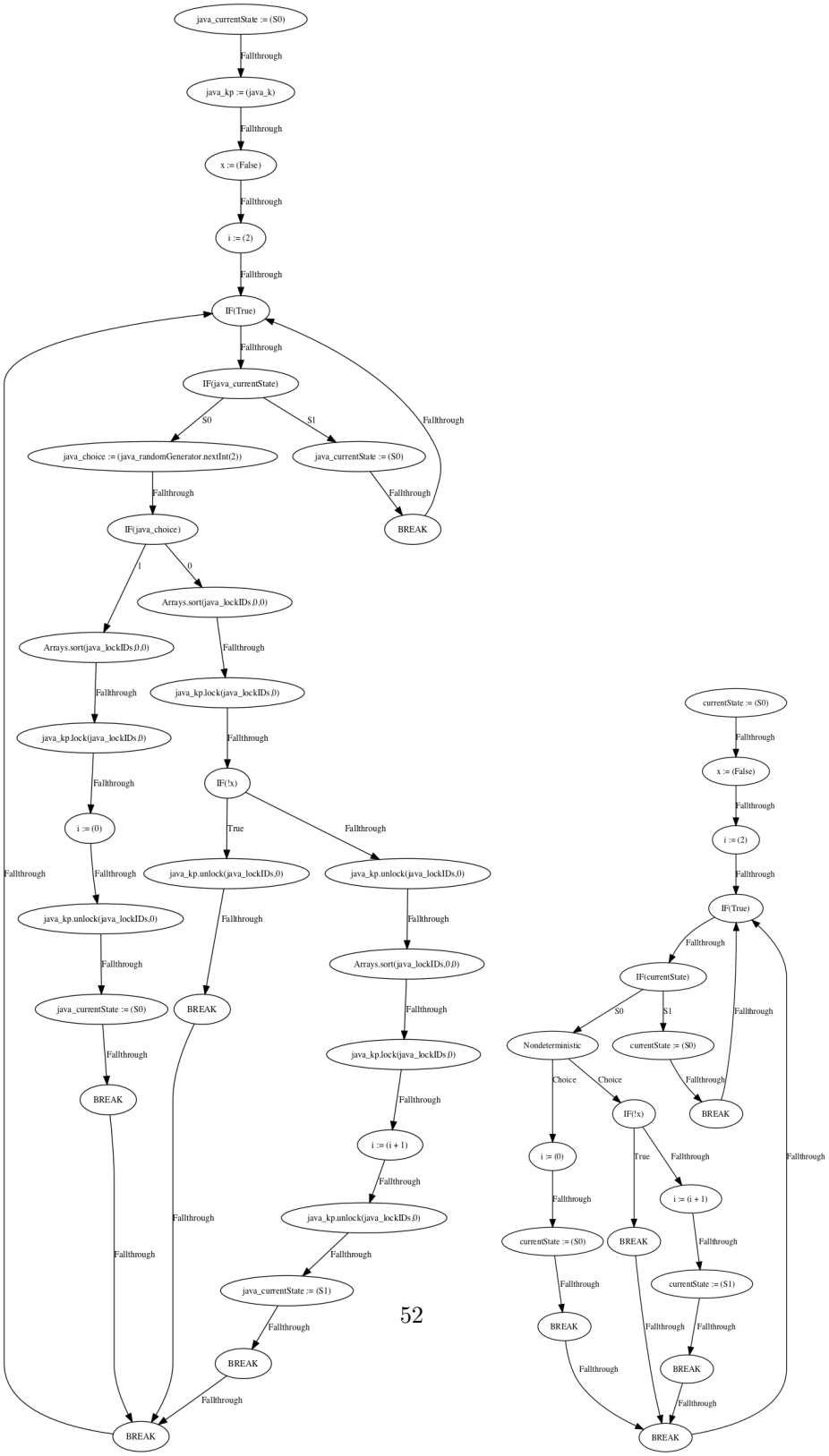


Figure 4.5: Removing Breaks

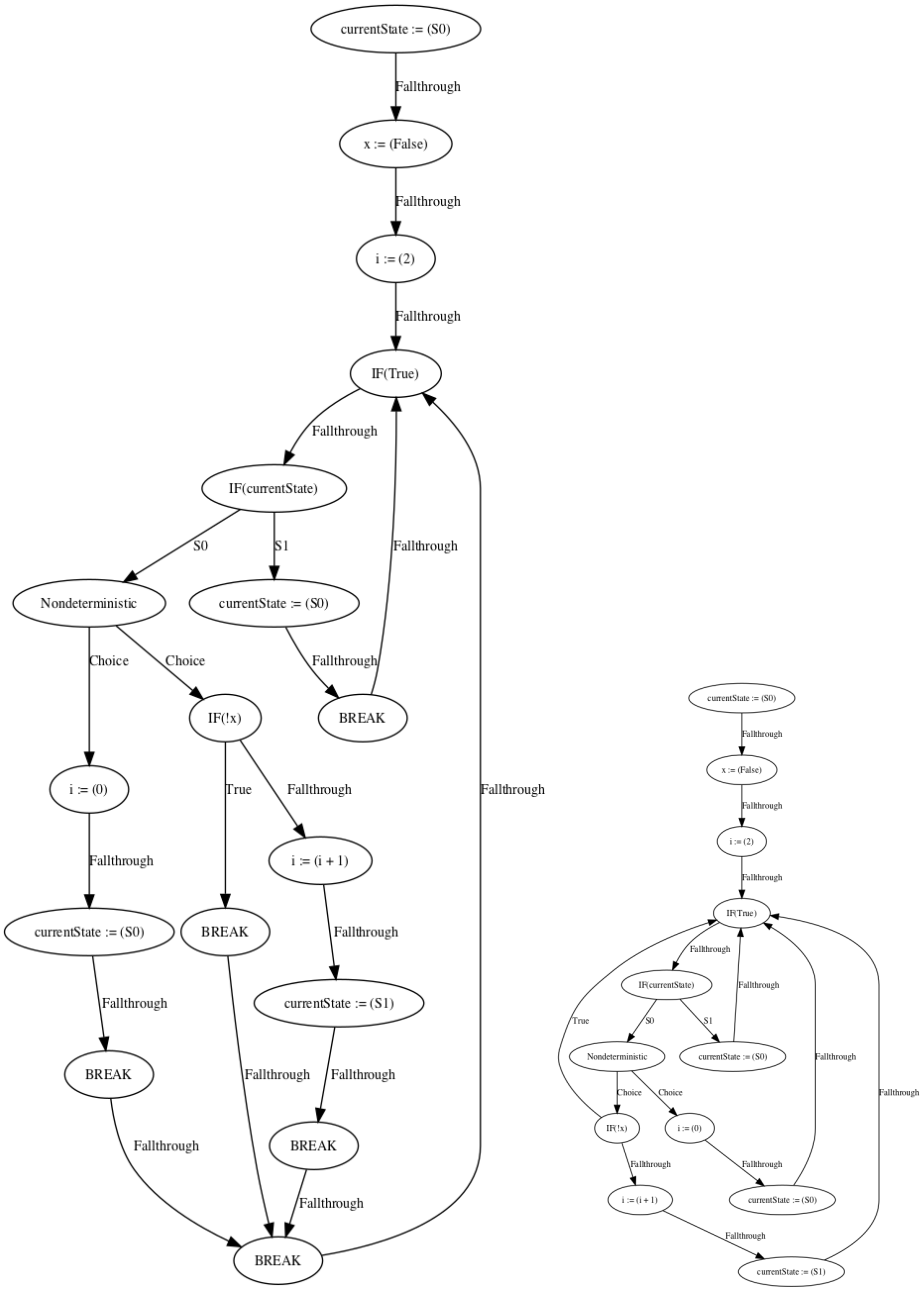
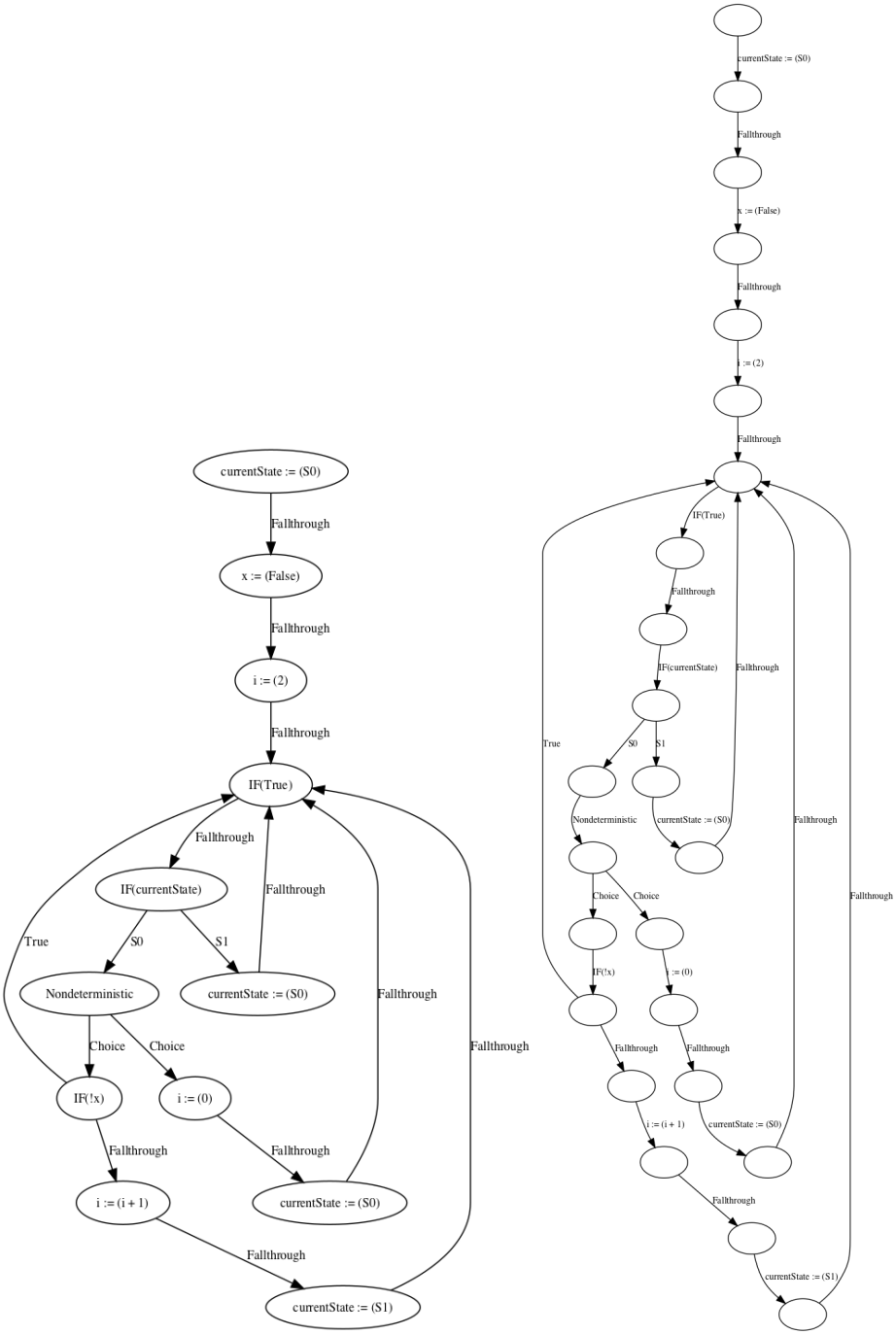


Figure 4.6: LTS Transformation



Chapter 5

Discussion

5.1 Correctness

To prove the correctness of our method we have to prove that for any SLCO model and its translation to Java the solution will give a correct answer to the question if they share the same behaviour. We split the argument into two propositions. First and foremost we have to argue that the generated Control Flow Graph of both sides is a valid and accurate representation of the program behaviour. Next we have to show that transforming the Control Flow Graphs to Labelled Transition Systems and checking their bisimilarity is a reliable method of proving that they are semantically equivalent.

5.1.1 SLCO Representation

We propose that the CFG generated from the SLCO model retains all the behaviour present in the original. To show that we start by defining the expectations.

We expect that:

1. The order in which the transitions may take place is preserved.
2. The transitions are enabled and disabled as according to the SLCO specification.
3. The initial state of the statemachine is preserved.
4. The variables in the state machine are initialized with the same values.
5. The change to the variables is happening in the same order and based on the same conditions as in the SLCO model.

6. No unexpected change to variables and behaviour altering the flow of the statemachine may happen.

Model

We extract from the model all the necessary information that is related to the execution. The main focus for us is which objects are created and what values are being assigned to the class variables in each object. By determining those facts we establish the starting state of the system and which statemachines will decide on how the program will change this state.

The CFGs we will use in the checking procedure are determined by which objects are initialized. For each initialized object we will generate Control Flow Graphs for all its statemachines. Each of those statemachines we will prepend with the sequence of assignments used to create the object.

To prove that this behaviour is in line with the SLCO semantics, we have to argue that:

1. It is correct to reason about the objects separately.
2. It is correct to reason about the statemachines separately.
3. The CFG representing the initial state of the system is accurate depiction of the SLCO initial state.
4. Generated CFGs are an accurate representation of the change in the system.

For the first statement, we look at how SLCO describes the interaction between the objects. According to the specification it is only possible for the objects to communicate through the channels. Because the current implementation of channels is missing from the SLCO translation framework we explicitly exclude it from our reasoning. If there is no interaction between objects then in fact we can examine them separately as they will not influence each other.

For the second statement, we have to know how the statemachines will interact with each other. SLCO semantics are oblivious of the notion of parallel execution. They are described in separately and the only means they can communicate through are the class variables that are shared between them. The values of those variables might influence the flow of the CFGs generated for those statemachines. We however describe the whole of the flow and do not care about any particular state of the system. The only problems that may arise with the communication are that of atomicity and simultaneously accessing shared variables, both of which we address in section 3.4.

For proving whether the initial state is being preserved we examine how the variable initialization is being translated. For each variable that is initialized

we generate an assignment node in the Control Flow Graph. The values are trivially being preserved and the order is being preserved as we sequence them in the same way the SLCO model does. This ensures that values of the variables will be exactly the same after the initialization, hence the initial state of the system is preserved.

Next we argue that the Control Flow Graphs generated for each statemachine are an accurate depiction of change in the system. An SLCO specification describes an execution of an object using statemachines. They are the only places except for object creation where values of variables may change. By translating all of them we ensure that all of the behaviour captured for an object will be present in the CFG. Furthermore the execution described by the statemachines follows the same level of abstraction as CFG does. Each transition will describe how the values stored in variables will evolve as opposed to an unique snapshot that will present their values at a certain point.

All these arguments lead to an observation that the initial state of the system is preserved with the translation and that the change of state will be accurately simulated. Essentially, by storing current values of variables and modifying them by traversing the graph, we should replicate the same behaviour as described by the SLCO model.

Statemachine

For each statemachine we generate a Control Flow Graph depicting the behaviour that it represents. To show that the translation is correct we need to prove the following:

1. The initial state and values of variables in a statemachine are preserved with the translation.
2. For any state A and B of the statemachine, if there exists a transition between them, there should be a sequence of actions in the CFG that represents the statements of this transition.
3. For any reachable state A in the statemachine, the statements for all possible transitions coming out of this state should be reachable in the Control Flow Graph.
4. If two states A and B are not joined with a transition there should be no direct flow between those states in the CFG.

These assumptions will in the end prove that the transitions may happen only if they are defined in the SLCO model. They will also prove that all of the transitions are translated. Furthermore, the order in which the transitions happen will be preserved. The correctness of translation of each of the statements

defined in the transition we leave for the next subsection and we concentrate on the general flow of the statemachine here instead.

SLCO statemachines do not explicitly store information about the current state, they do however specify the initial state. Leaving the information about the current state implicit could in turn result in the Control Flow Graph being infinitely long as we would have to translate all the combinations of transitions that may happen. To solve this issue we introduce a variable that will store the information about the current state. By assigning the initial state to this variable and translating the statemachine variable initialization to a sequence of assignments we ensure that the initial state is preserved.

In the CFG translation, all of the transitions are being queried for the state that is the beginning point of the transition. The set of these states is used as a switch on which behaviour should be executed. From there, depending on the number of outgoing transitions from a state, the CFG will behave in one of two ways:

1. If there is only one ongoing transition, the statements of it will be translated one by one and connected in sequence.
2. If there is more than one ongoing transition, we introduce an nondeterministic choice condition. The CFG will branch to multiple cases depicting the transitions and translate their statements into those cases.

After completing the transition statements, we assign to the current state variable the value of the target state of that transition.

We split our reasoning depending on the out transition from a state. In case there is only one outgoing transition, by assigning the initial state to a variable and checking that variable we immediately arrive at the sequence of nodes representing the statements in that transition. In this case the second assumption holds for state A.

In case there is more than one outgoing transition, the first condition filters which transitions can be executed and the nondeterministic choice ensures that the statements of one of them can be reached. In this case the second assumption also holds for state A.

By assigning the target state of the taken transition to the current state variable we ensure that the assumption holds for all states in the system. The system after completing the transition reverts to the main loop which in turn leads to the condition checking the current state. We can continue this process as long as there are transitions leading from the current state. Hence for any state in the system that is reachable we can execute the statements of the transitions coming out of it.

For the third assumption we look into the nondeterministic condition node. When reaching this node in CFG, the system will choose an arbitrary branch and follow it. This means that any of the branches representing outgoing transitions are reachable. It may happen that the system upon reaching the condition will always choose a certain transition which would make it impossible to execute statements of the other. SLCO does not however provide any promises with regards to fairness. The transitions are possible to be taken but as it is an nondeterministic choice, they do not have to be.

While fairness is not addressed by the SLCO specification it can greatly influence the quality of the translated code. In her thesis Zhang[33] elaborates on the topic. It can happen that one of the threads representing a statemachine will continuously grab the CPU time. As a result other threads in the system would starve, which goes against the intended meaning. In her solution she uses fairness policies implemented for some of the standard library classes and eliminates the problem of starvation.

The last assumption again follows from the fact that we re-assign the variable holding the current state at the end of each transition. For the direct flow to happen between two states that are not connected the value assigned to this variable would have to be different than what the transition specifies. Since we are generating the control flow on transition basis we exclude that possibility.

Statement

In the previous subsection we have shown that the statements are being translated in order in which they are defined in the SLCO model. We thus have to concentrate on the meaning of the particular statement to the system. Since SLCO translation framework supports only three types of statements at this moment, we concentrate on those.

Assignment is being directly translated to an assignment node. As they have the exact same meaning the behaviour is preserved.

Expression statement, in case it is the first statement in the transition, is translated to a condition that checks the expression body and will revert to the main loop if needed. If it is not the first it is translated to a loop which continuously checks the expression body until it evaluates to true. In SLCO the expression is used to either enable the transition or to hold it until the condition is satisfied. As it may never happen that the state of variables is partially changed by the transition we have to verify that the translation does preserve that property. In the first case the CFG immediately reverts to main loop if the condition is not satisfied. Since no other statement was executed,

the behaviour is preserved. In the other case, the loop will ensure we do not proceed with the execution until the expression evaluates to true. This will make it impossible to execute any other statements and ensure the system is in correct state. This proves that the semantics of this statement are being preserved.

The composite statement after decoupling the atomicity property has no special meaning and behaves exactly the same as the assignments that are being directly translated in order. In case there is a guarding condition on the statement, it behaves the same as the expression condition hence still preserves the meaning.

5.1.2 Java

The Java translation to CFG is very much straightforward. We traverse the AST while simultaneously building the graph. In essence all the conditional constructs - `if`, `if else`, `switch` get translated to a conditional node. The loop instructions `while`, `for` get translated to a conditional node that will return to check the condition after the clause is complete. All the assignments are translated directly and they retain the same behaviour. Method invocations are missing from the specification of SLCO and we expand them where applicable to the statements they contain. We include the method calls we cannot expand as effect nodes to ensure no unknown behavior happens in the system. By leaving those nodes inside the processed CFG we make sure the check will fail as they will not have their counterparts in the graph generated from SLCO.

5.1.3 Optimizations

We have argued that the generated Control Flow Graphs are preserving the behavior of the SLCO model and Java code. Because we optimize the CFG before the check, we also need to prove that the optimizations will not have any negative effect on the test.

Removing Dead Code

The first optimization is done for the Java translation before the generation of the Control Flow Graph. The changes introduced will however still carry to the CFG. As we are removing code that would later result in additional branches in the graph, we have to ensure that the removal is justified and does not change the behavior.

What we have to show is that the removed branches are unreachable based on all possible variable assignments in the CFG. The assignments are being propagated symbolically and we do not evaluate the values of expressions. This would pose a problem as some values could be incremented and then checked against a specific value. This could potentially happen after many iterations of the program. We therefore restrict ourselves to checking the value of the current state.

In the Java generated code, the value of the current state variable is defined as an enum. What it means in practice, is that the traditional arithmetics will not apply to the value. It is still however possible to cast the the value to an integer type and modify it. As the only way to check whether the casted value will be changed to a different state is to evaluate every expression, we avoid it by manually ensuring that the Java translation does not introduce any enum manipulation. With the assurance that current state will always be assigned a concrete value we can safely compute a set of all possible values. With the set in place we can definitely tell which branch will never happen and therefore is not part of the behavior description.

Java Refactorings

To prove that the graph refactorings specific to Java are not changing the behavior we have to show the following:

1. Removing `java_` prefixes does not influence the behavior.
2. Removing the usage of `Keeper` class does not influence the behavior.
3. Generating a random integer and branching on its value is equivalent to a nondeterministic choice.

For the first statement, we identify two cases in which the behavior might change after applying the optimization. The first possibility is that a variable in the original SLCO model was named using the `java_` prefix. We find this not to be a problem as it is highly unlikely that it happens. On some rare occasion when it does happen, the translation to Java will append the prefix before the variable name anyway. The optimization will only remove the first prefix, hence the behavior will be preserved. This can also be checked using a simple string search of an SLCO textual representation. The other possibility is that removing the prefix will create a conflict in the naming, effectively changing a prefixed variable to one that is already defined. Since the translation process will prefix only a handful of variables this can also be solved by a searching the occurrence of those variables in the model.

To argue that the second assumption does not change the behavior, we have

to look into the implementation of the `Keeper` class. To ensure that removing the uses of the class will not change the behavior we have to safeguard two important properties. The class cannot modify the values of the variables in the system and it cannot change the flow of the program. By inspecting the class body we conclude that the only variables accessed inside its methods are that encapsulated in the class itself. Therefore it cannot change any external values. The class also provides only two methods - `lock` and `unlock` which will lock or unlock variable locks inside the class. We argued previously that the atomicity properties are preserved, hence we do not need to verify that. There are no instructions that might change the flow of the program inside those methods. Since the class is always generated in the same manner and does not change with every translation of the SLCO model, it is enough to only inspect it once and conclude that it is safe to remove it.

The last part of the Java specific refactorings is the replacement of the Java nondeterministic choice implementation with an nondeterministic node in the CFG. To prove that it is correct to do so we have to argue that the implementation correctly describes the nondeterministic choice. The implementation relies on the use of `Random` class. Looking into the description of the class in Java documentation[22] we find that the class will generate a pseudorandom set of uniformly distributed numbers. We use those numbers to decide which branch will be chosen in the choice condition. As the distribution is uniform it is equally likely to choose any of the branches. This in turn means that the choice is indeed nondeterministic and we can replace the whole mechanism with just the nondeterministic branching node.

Removing Breaks

To argue that removing `Break` nodes from the Control Flow Graph does not change the behavior of the system we investigate the semantics of the `Break` node. The node was used in the IR format to mark that the successor of the node should be the last visited `Switch`. After translation to CFG the node no longer has any meaning as it does not change the state of the variables nor it changes the flow of the application. Because of that it can be completely ignored and thus can be removed from the graph.

5.1.4 Translation to LTS

Last step of the translation is to transform the obtained CFG to a Labelled Transition System. Similarly to the previous steps we have to argue that the

translation will preserve the starting state of the system, that the changes of the system will have the same effect and that they are executed in the same order.

We can look at the translation procedure as moving all of the information from nodes to the edges. Since we do not replace any information stored on the nodes but rather remove it, we do not introduce any new operations to the system. This means that we never change the meaning of particular operations described on the CFG. Hence the second statement is trivially true.

To show that the order of execution is preserved we look at the shape of a single node after translation. For each of the nodes in the system we introduce a new edge that will connect the incoming and outgoing edges of the node. This means that the order of execution of those edges is preserved. Furthermore since the introduced edge does not connect to any other, it cannot alter the flow of the program, which means the second assumption is correct.

To see how the system state is initialized we look into the beginning of the graph. The initialization of the variables should be described by a sequence of assignments. This sequence will always be uninterrupted as there is no conditional flow needed for the system initialization. If we neither introduce nor remove any information and the sequence order is preserved, the initial state will be preserved in the translation. Based on two previous assumptions we conclude that this is indeed the case.

5.2 Testing

To test that the procedure in a real scenario, we have applied the `CFG-Tool` to a number of SLCO models along with its translations to Java. The SLCO models were obtained by translating the BEEM benchmark models[23] to the SLCO format. We then have run the SLCO 2.0 translation framework on those models and stored them together on disk. The dataset has been chosen as it contains models of well known problems and with established properties. It spans multiple domains and contains different scales of the problems so the testing can be done on more complicated examples. Table 5.1 presents a list of used models with short descriptions of what problems they represent.

We execute the `CFG-Tool` on all of the models and present the results along with the time needed to generate the answer in table 5.2.

As it can be seen from the table all of the translations from SLCO model to Java source code will represent the same behavior. This was expected as the translation is quite straightforward and the Java source code is already quite

Table 5.1: Used BEEM model descriptions

Model Name	Model Description
adding	Concurrent adding puzzle
anderson	Anderson queue lock mutual exclusion algorithm
bakery	Bakery mutual exclusion algorithm
driving_phils	Mutual exclusion of processes accessing several resources
elevator_planning	Planning of elevator strategy under several constraints
elevator	Elevator controller
exit	Model of a city team game
frogs	2D Toads and Frogs puzzle
lamport	Lamport fast mutual exclusion algorithm
leader_filters	Leader election algorithm based on filters
loyd	Sam Lloyd fifteen puzzle
mcs	MCS queue lock mutual exclusion algorithm
msmie	Multiprocessor Shared-Memory Information Exchange protocol
peg_solitaire	Peg solitaire, an old board game for one player
peterson	Peterson mutual exclusion protocol for N processes
phils	Dining philosophers problem
rushhour	A sliding block puzzle
schedule_world	Scheduling of machines for production
sokoban	Sokoban sliding block puzzle
telephony	Telecommunication service

Table 5.2: Result of the check

Model Name	Answer	Time(s)
adding.1	True	0.18
adding.2	True	0.17
adding.3	True	0.17
anderson.1	True	0.17
anderson.2	True	0.18
anderson.3	True	0.2
anderson.4	True	0.22
anderson.5	True	0.2
anderson.6	True	0.22
anderson.8	True	0.24
bakery.1	True	0.22
bakery.3	True	0.22
bakery.5	True	0.25
bakery.7	True	0.24
driving_phils.1	True	0.27
driving_phils.2	True	0.28
driving_phils.3	True	0.27
driving_phils.4	True	0.29
driving_phils.5	True	0.55
elevator2.1	True	0.21
elevator2.2	True	0.3
elevator2.3	True	0.25
elevator_planning.1	True	0.25
elevator_planning.2	True	0.25
elevator_planning.3	True	0.2
exit.1	True	0.25
exit.2	True	0.47
exit.3	True	0.4
exit.4	True	0.48
exit.5	True	1.03
frogs.1	True	0.28
frogs.2	True	0.24
frogs.3	True	0.37
frogs.4	True	0.28
frogs.5	True	0.42
lamport.1	True	0.26
lamport.2	True	0.28
lamport.3	True	0.24
lamport.5	True	0.29
lamport.6	True	0.25
lamport.7	True	0.31
lamport.8	True	0.28
leader_filters.1	True	0.21
leader_filters.2	True	0.2
leader_filters.3	True	0.26
leader_filters.4	True	0.21
leader_filters.5	True	0.25
leader_filters.6	True	0.25
leader_filters.7	True	0.36
loyd.1	True	0.19
loyd.2	True	0.25
loyd.3	True	0.22
mcs.1	True	0.24
mcs.2	True	0.26
mcs.3	True	0.25
mcs.4	True	0.25
mcs.5	True	0.25
mcs.6	True	0.33
msmie.1	True	0.3
msmie.2	True	0.46
msmie.3	True	0.58
msmie.4	True	0.97
peg_solitaire.1	True	0.35
peg_solitaire.2	True	0.85
peg_solitaire.3	True	1.4
peg_solitaire.4	True	0.38
peg_solitaire.5	True	0.81
peg_solitaire.6	True	0.52
peterson.1	True	0.21
peterson.2	True	0.2
peterson.3	True	0.2
peterson.4	True	0.19
peterson.5	True	0.24
peterson.6	True	0.25
peterson.7	True	0.22
peterson_simple	True	0.17
phils.1	True	0.19
phils.2	True	0.21
phils.3	True	0.25
phils.4	True	0.26
phils.5	True	0.3
phils.6	True	0.47
phils.7	True	0.34
phils.8	True	0.36
rushhour.1	True	0.26
rushhour.2	True	0.28
rushhour.3	True	0.34
rushhour.4	True	0.27
schedule_world.1	True	0.25
schedule_world.2	True	0.32
schedule_world.3	True	0.33
sokoban.1	True	0.2
sokoban.2	True	0.19
sokoban.3	True	0.19
telephony.1	True	0.29
telephony.2	True	0.33
telephony.3	True	0.34
telephony.4	True	0.35
telephony.5	True	0.45
telephony.6	True	0.43
telephony.7	True	0.41
telephony.8	True	0.45

similar in form to the Control Flow Graph. Since we have argued correctness of the translation from SLCO to the CFG, it is not surprising to see that the result of the check is true for all models.

The checking procedure has a reasonable performance. Even when running the application on some of the more complicated samples, the time needed to finish the check is well within an acceptable range of around a second. It has to be noted that the performance was not the main goal of the implementation and thus no special effort has been made to increase it. What was important at this stage of the project was to ensure that the procedure behaves as expected and that it does veritably answer the question of semantics equivalence.

Important thing to recognize is that the project is designed to be used as part of the building process. Because quick feedback is important in the development process and every fraction of a second can make a significant difference, at a later stage it might be useful to re-visit the performance of the implementation.

To exclude the possibility of a false positive we introduce an error into how the translation proceeds. By deliberately breaking the code generation we expect to show that the check will report a problem if the semantics of two system do not match.

We intend to introduce three modifications into how the code is translated:

1. Change the initialization of the variables to show that the system needs to preserve the same starting state.
2. Remove the conditions from execution of the transitions to show that they need to be preserved.
3. Change the order in which the transitions are executed to show that its order has to be preserved.

After modifying the Java translation and regenerating the source code we rerun our test suite as presented on table 5.3.

5.2.1 Variable values

We modify the Java translation not to translate any variable initialization and instead assign the default value according to the SLCO semantics. The preservation of the starting state of the system is a crucial step in our testing process. The checking cannot rely only on the change to the variables. Having a different state of variables at the start of the system would break some of the functional properties that were checked on the original model.

In column 1, table 5.3 presents the results of re-applying the check on the broken translation. It immediately becomes apparent that some of the results

Table 5.3: Results of the check of broken translations

Name	1	2	3
adding.1	False	False	False
adding.2	False	False	False
adding.3	False	False	False
anderson.1	False	False	False
anderson.2	False	False	False
anderson.3	False	False	False
anderson.4	False	False	False
anderson.5	False	False	False
anderson.6	False	False	False
anderson.8	False	False	False
bakery.1	True	False	False
bakery.3	True	False	False
bakery.5	True	False	False
bakery.7	True	False	False
driving_phils.1	False	False	False
driving_phils.2	False	False	False
driving_phils.3	False	False	False
driving_phils.4	False	False	False
driving_phils.5	False	False	False
elevator2.1	True	False	False
elevator2.2	True	False	False
elevator2.3	True	False	False
elevator_planning.1	False	False	False
elevator_planning.2	False	False	False
elevator_planning.3	False	False	False
exit.1	True	False	False
exit.2	True	False	False
exit.3	True	False	False
exit.4	True	False	False
exit.5	True	False	False
frogs.1	False	False	False
frogs.2	False	False	False
frogs.3	False	False	False
frogs.4	False	False	False
frogs.5	False	False	False
lamport.1	False	False	False
lamport.2	False	False	False
lamport.3	False	False	False
lamport.5	False	False	False
lamport.6	False	False	False
lamport.7	False	False	False
lamport.8	False	False	False
leader_filters.1	True	False	False
leader_filters.2	True	False	False
leader_filters.3	True	False	False
leader_filters.4	True	False	False
leader_filters.5	True	False	False
leader_filters.6	True	False	False
leader_filters.7	True	False	False
loyd.1	False	False	False
loyd.2	False	False	False
loyd.3	False	False	False
mcs.1	False	False	False
mcs.2	False	False	False
mcs.3	False	False	False
mcs.4	False	False	False
mcs.5	False	False	False
mcs.6	False	False	False
msmie.1	False	False	False
msmie.2	False	False	False
msmie.3	False	False	False
msmie.4	False	False	False
peg_solitaire.1	False	True	False
peg_solitaire.2	False	True	False
peg_solitaire.3	False	True	False
peg_solitaire.4	False	True	False
peg_solitaire.5	False	True	False
peg_solitaire.6	False	True	False
peterson.1	True	False	False
peterson.2	True	False	False
peterson.3	True	False	False
peterson.4	True	False	False
peterson.5	True	False	False
peterson.6	True	False	False
peterson.7	True	False	False
peterson_simple	True	False	False
phils.1	True	False	False
phils.2	True	False	False
phils.3	True	False	False
phils.4	True	False	False
phils.5	True	False	False
phils.6	True	False	False
phils.7	True	False	False
phils.8	True	False	False
rushhour.1	False	False	False
rushhour.2	False	False	False
rushhour.3	False	False	False
rushhour.4	False	False	False
schedule_world.1	False	False	False
schedule_world.2	False	False	False
schedule_world.3	False	False	False
sokoban.1	False	True	False
sokoban.2	False	True	False
sokoban.3	False	True	False
telephony.1	False	False	False
telephony.2	False	False	False
telephony.3	False	False	False
telephony.4	False	False	False
telephony.5	False	False	False
telephony.6	False	False	False
telephony.7	False	False	False
telephony.8	False	False	False

did not change compared to the original check. We investigate all of the models that returned a positive answer and conclude that the result is according to the expectation. Models that returned the `True` answer either had the variables initialized with the same values as the SLCO defaults or they did not initialize the values at all.

5.2.2 Expression removal

We modify the Java translation to ignore all the expression statements in the transitions. Translating conditional flow of the program is crucial to preserve its semantics and the satisfiability of functional properties. The results are presented in table 5.3 in column 2. Similarly to the first change, we can observe some `True` results after this breaking change. The positive results are given for models `peg_solitaire` and `sokoban`. After examining the models we conclude that they do contain conditional execution. In the translation however we removed handling only of the expression nodes, while it is also possible for the condition to be expressed as a guard to a composite statement. In the mentioned models all of the conditions are defined in this manner and hence are being properly translated to the Java equivalent.

5.2.3 Transition ordering

In each transition we reverse the order to make the original source state the target and the original target state the source. Since reversing the flow should completely change the meaning of the resulting graph, unless it was symmetrical, for example contained a single transition ending up in the same state, we expect all of the models to break. The results are exactly as we anticipate. We double check the test set of models for symmetry and conclude that the property does not hold in any.

5.2.4 Conclusion

The results before and after modifying the translation process show that the change of behavior does make an important difference and is properly tested by the `CFG-Tool` procedure. By verifying whether the Control Flow Graphs are the same in both the original model and the translation we can conveniently check if both sides of the system represent the same semantics. If they do this means that any property that holds in the original model will also hold in the generated code.

5.3 Limitations

The presented solution to the problem of checking whether the translation from SLCO model to Java source code does not come without problems.

The main limitation of the solution is that it does not prove the semantics preservation for every possible translation but rather for a specific model and the code generated from it. User has to make sure that the check is performed every time the model is modified and the code is re-generated. Unfortunately verifying the complete semantics preservation for a given translation is a complicated task and involves extensive proofs as described in [19]. In our solution, we have tried to minimize the inconvenience by ensuring that the check can be easily incorporated into the development pipeline. The time needed to obtain the result is short enough for it to be a part of a build procedure. Additionally, the application will require only one simple command and will perform the procedure automatically without user intervention.

Another problem arises from the step of developing the translation to the IR. It is obligatory that the developer of such translation has a good understanding of the original language semantics and that they can be expressed using the format. For the latter it is not much of an issue - if a model cannot be expressed in terms of executable code then there is little interest in translating it to Java source code. For the former however it is crucial that the translation is error free and that the details are well defined. While the presented application tries to simplify the process as much as possible it is still a tedious manual labor to write the translation functions and ensure they are correct.

Finally, tangential to the previous problem is that the translated Control Flow Graphs do not allow for much of freedom. The presented check is very rigid and some behaviors that might in fact be equivalent in terms of semantics are not allowed to differ from the CFG generated from the other side.

An example of such behavior would be the case when the first statement of a transition is an expression. If the condition for transition is not satisfied, translation to CFG ensures that the program will return to the main loop node. This is not strictly needed however. The expression statement cannot change the value stored in current state. Therefore a more natural behavior in this case would be to return to the choice of transition, that are coming out of the current state - the nondeterministic choice node. Although the current translation is still correct, the user has to keep track of what behavior is equivalent as it may lead to false negatives. Furthermore it nearly eliminates the possibility to develop the translation to CFG for each language separately - it is very likely

the translation to CFG would be influenced be the way model is translated into code. This lack of separation makes the solution less general and can even lead to carelessness when it comes to defining what behavior is correct when translating the model.

5.4 Related Work

There has been some work put into proving that a model transformation is preserving the semantics. In [13] authors show that by defining the transformation as a triple graph grammar, we can check if the relation between the input and output model show a bisimulation relation. If the respective graphs are indeed bisimilar it would mean that transformation preserved the semantics and the behaviour of both is equivalent. The other technique described by the authors is to show that the left and right hand sides with certain interface are for every rule bisimilar with respect to the operational rules. The disadvantage of the solutions given, are that both input and output model has to exist on the same level of abstraction and be a runnable program. Alternatively they both have to have specified operational semantics on which we can draw conclusions. Another disadvantage is that we need to always specify the transformation as three graphs - the input, the output and the mapping.

In comparison to [13] our approach does not rely on any particular shape of the original language. The languages and the transformation can be specified in any preferable manner. It does require more effort to write the translation to CFG but is more general and can be used in many scenarios not taken into account in [13].

In [1] Ab Rahim and Whittle take a slightly different approach. They argue that verifying the end product of a transformation is much more practical than verifying the transformations themselves. By using a technique they call an Annotation-Driven Model Checking (ADMC) they aim to check if the result's semantics match those of the original model. They start off by manually describing the semantics of UML state machine diagram as a list of properties such as entry order, exit order, entry via entry points, exit via exit points and others. For each of those properties a specific assertion has to be developed in the source language along with a transformation to the Java source code resulting from the transformation. After this is done, the code is being run via the Java Path Finder to verify all of those properties still hold after the transformation. The obvious advantage of this approach is that the assertions have to only be defined once per input model language. The end user, also called consumer,

is only responsible for running Pathfinder to check if the properties still hold in the end model. The other advantage is that the transformation language is treated as a black box in this scenario, no access to the tool's source code is needed as everything is verified on the resulting Java code. The shortcomings of ADMC are that it only verifies the properties verified by the authors and does not prove full semantic preservation between the two models. The other disadvantage is that the semantic conformance is proven only for the one particular instance of the model and not all possible transformations in general. It also requires some effort from the end user as the verification has to be run each time the model or the transformation is modified. Semantic preservation is also important in the domain of compilers which can be looked at as a model transformation from source code to assembly. Several work has been published with regards to proving that compilers are bug free and do not change the behaviour of the program.

What our solution does in contrast to the paper by Ab Rahim and Whittle is the fact that we do not concentrate on any particular property of the system. Our checking process does not take properties as input and instead opts to show that all possible properties will be preserved by the translation. By focusing on the semantics preservation we remove the need to prove any particular property and rely on the fact that they were holding in the original model. We also do not restrict our method to any particular language and hence remove the reliance on Java Path Finder. Similarly to our solution, the one described in the paper has to be run each time the transformation is done.

In [19] author verify semantic preservation in a compiler of a subset of C language called CLight. The verification is done by sequentially translating the source into 8 different intermediate languages. On each stage some original complexity is removed and the code is moved closer to the target abstraction level of Power PC assembly language. Assurance of behaviour preservation comes from the fact that each of those languages have a full semantic specification. Each transformation comes with a set of Coq proofs based on functions and pattern matching, inductive or coinductive predicates representing inference rules and by predicates in first order logic. The most important finding of the paper is that it is possible to prove correctness of a compiler using proof assistants and make sure that it will produce correct code to at least assembly level. The obvious problem with the solution is its workload - the author estimates developing of the proof took 3 man-years and produced 42000 lines of Coq code. This however is only one time work and the proof does not need to be rewritten if the language does not change. Furthermore the performance of the end product

is comparable with those currently on the market.

In comparison to the work in [19] our solution to the problem is far less work intensive. Unfortunately if one wanted to extend the solution described in the paper by another language, the majority of the process would likely have to be repeated. The time needed to design the original solution would make it very difficult to do. It does represent however a more complete proof of semantics preservation than the work we describe in this thesis. Once developed the check proves the semantics preservation for every single instance of translated model whereas our solution has to be redone for every translation. It has to be noted however that the original solution is done on a toy language with simplified semantics and is not likely to be used outside its specific domain.

Kumar et al [17], develop a verified compiler for a subset of ML functional language - CakeML. The benefit of their solution is that they can now verify the translation up to the machine code which was missing in [19]. Furthermore they are the first to build a formally verified compiler which bootstraps itself, meaning that the compiler for CakeML is then rewritten in CakeML which proves that no bugs are present.

Compared to the work of Kumar et al, our solution is more universal in terms of the languages that can be verified. In the case presented by the paper the compiler for a single language is verified while we concentrate on making the tool available for multiple transformation type. Where the solution presented in the paper excels however, is the fact that it can be used to verify itself. Our solution requires significant amount of labor to show that the transformation to CFG indeed is correct and some errors can still occur in the code used to translate the model. By providing a way of automatically checking itself this method would be resistant to these problems.

Chapter 6

Conclusions

After discussing the correctness and the assessment of the project we can revisit the research questions stated in the introduction. We start by answering the smaller questions to finally answer the main goal of the thesis.

RQ_1: *How can you establish that the end product of model transformation preserves functional properties of the input?*

We have showed that for models that are designed to be executed on a real world computer, we have to examine their execution from the machine perspective. By establishing the state of the variables at the start of the system and by determining how this state may change we can describe the total of behavior of the system. By extracting this information from both model and the generated code we can use the bisimilarity relations to verify that at any point if it is possible to perform an action in the model there should be such possibility in the corresponding code. If this is indeed the case that means that the systems are semantically equivalent and they can never diverge in their behavior. This in turn results in the fact that any possible property will be preserved with the translation.

RQ_2: *How can you bridge the semantic gap between the transformed languages?*

The problem with the semantic gap that we have investigated in the thesis is that very often the information that is missing from the description cannot be automatically filled. Unfortunately the interpretation of the specific behavior of certain language constructs has to be manually done by the developer. By providing a common representation that will contain concepts from both sides of the translation, we can ensure that it is possible to properly map the constructs between the abstraction barrier. By choosing the most detailed semantics we

can ensure that even the smallest detail will be preserved between the model and the code.

RQ_3: *How can we automate the checking process, so it can be done with minimal user effort?*

As mentioned, we cannot provide a fully automatic check that will be able to run with no user effort at all. We can however minimize the needed attention. Once the translation to the common IR is described by the translation developer it is possible to just run the process on the original model and the code generated from it. Unfortunately it has to be done every time the model is changed and if the translation from model to code is modified. In most cases however the check takes little time and can be done as part of the build process. The task of running the process was simplified to one simple command that will take the source model and the generated code and will provide a definitive yes or no answer. Assuming the developer of translation does put effort into ensuring that the translation to IR is in place as well, the binary application ensuring the equivalence can be shipped with the translation binary.

We return to the main research question of the thesis:

RQ: *How can you verify that the generated code is correct with regards to the source model?*

Given a model and the code generated from it. By bringing both sides of the equation to a common denominator represented by the Control Flow Graph, we ensure that the semantics will be the same of both SLCO model and Java source code. By verifying that there exists a bisimilarity relationship between Control Flow Graphs representing the model and the code, we ensure that the beginning state and the change of the state of the system will be identical for both at any point of the execution. This in turn results in the state of the system being identical in those points. As there is no possibility of a different path of execution we conclude that the model and the generated code are semantically equivalent. If they are semantically equivalent then any functional property that holds in the source model also holds in the resulting code.

6.1 Future Work

The method described in this thesis can be extended to any model-to-code translation in a straightforward manner. Furthermore it is not necessary to restrict the target of translation to the executable code as long as it describes a sequence of actions in a system. The SLCO translation framework supports mCRL2 formulas as the generator output, it should be possible to check the

translation using the process described in this thesis.

During the development of the project we rely on results described in [33] for atomicity preservation. It should be possible to encode the guarantees of atomicity into the Control Flow Graph. By making it explicit in the graph, what parts of the program should represent an atomic step in computation, we could verify that the atomicity is preserved. This would also enable checking the property in other languages for which the preservation has not been proven yet.

The work can also be extended by providing formal semantics to the IR code to ensure that the translation of certain structures will preserve the desired meaning. By verifying that the semantics of the structures found in the original model strictly match the ones of the generated IR we would eliminate the chance of erroneous translation to CFG.

Additionally, describing IR in terms of formal semantics could result in a verified method for automatically translating to IR constructs of both source model and target language of the transformation. Assuming the language of the model and the IR have a formal description of meaning, each construct could be mapped to a specific concept in their counterpart. The CFG check would then ensure that the order of execution flow between the constructs is preserved.

Finally, the main disadvantage of our solution is that it can only verify semantics preservation for an instance of a model and the source code generated from it. Further research may include generalizing the technique to verify the translation itself and remove the need to re-apply the tool after every modification of the model.

Bibliography

[1] Lukman Ab Rahim and Jon Whittle. "Verifying Semantic Conformance of State Machine-to-Java Code Generators". In: *Model Driven Engineering Languages and Systems*. Ed. by Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 166–180. ISBN: 978-3-642-16145-2.

[2] Frances E Allen. "Control flow analysis". In: *ACM Sigplan Notices*. Vol. 5. 7. ACM. 1970, pp. 1–19.

[3] Paul Baker, Shiou Loh, and Frank Weil. "Model-Driven Engineering in a Large Industrial Context — Motorola Case Study". In: *Model Driven Engineering Languages and Systems*. Ed. by Lionel Briand and Clay Williams. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 476–491. ISBN: 978-3-540-32057-9.

[4] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. "Precise Semantics of EMF Model Transformations by Graph Transformation". In: *Model Driven Engineering Languages and Systems*. Ed. by Krzysztof Czarnecki et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 53–67. ISBN: 978-3-540-87875-9.

[5] Cristiano Calcagno and Dino Distefano. "Infer: An automatic program verifier for memory safety of C programs". In: *NASA Formal Methods Symposium*. Springer. 2011, pp. 459–465.

[6] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[7] Sjoerd Cranen et al. "An overview of the mCRL2 toolset and its recent advances". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2013, pp. 199–213.

[8] Alberto Rodrigues Da Silva. "Model-driven engineering: A survey supported by the unified conceptual model". In: *Computer Languages, Systems & Structures* 43 (2015), pp. 139–155.

[9] LJP Engelen. "From Napkin sketches to reliable software". In: (2012).

[10] Martin Erwig. "Inductive graphs and functional graph algorithms". In: *Journal of Functional Programming* 11.5 (2001), pp. 467–492.

[11] Jean-Claude Fernandez et al. "CADP a protocol validation and verification toolbox". In: *International Conference on Computer Aided Verification*. Springer. 1996, pp. 437–440.

[12] Jan Friso Groote and Mohammad Reza Mousavi. *Modeling and analysis of communicating systems*. MIT press, 2014.

[13] Mathias Hülsbusch et al. "Showing Full Semantics Preservation in Model Transformation - A Comparison of Techniques". In: *Integrated Formal Methods*. Ed. by Dominique Méry and Stephan Merz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 183–198. ISBN: 978-3-642-16265-7.

[14] Graham Hutton. "Higher-order functions for parsing". In: *Journal of functional programming* 2.3 (1992), pp. 323–343.

[15] Graham Hutton and Erik Meijer. "Monadic parsing in Haskell". In: *Journal of functional programming* 8.4 (1998), pp. 437–444.

[16] Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[17] Ramana Kumar et al. "CakeML: A Verified Implementation of ML". In: *SIGPLAN Not.* 49.1 (Jan. 2014), pp. 179–191. ISSN: 0362-1340. DOI: 10.1145/2578855.2535841. URL: http://doi.acm.org/10.1145/2578855.2535841.

[18] *Language Java Haskell Module*. https://web.archive.org/web/20170724004425/https://hackage.haskell.org/package/language-java. Accessed: 2018-10-21.

[19] Xavier Leroy. "Formal Verification of a Realistic Compiler". In: *Commun. ACM* 52.7 (July 2009), pp. 107–115. ISSN: 0001-0782. DOI: 10.1145/1538788.1538814. URL: http://doi.acm.org/10.1145/1538788.1538814.

[20] Steve McConnell. *Code Complete, Second Edition*. Redmond, WA, USA: Microsoft Press, 2004. ISBN: 0735619670. URL: http://portal.acm.org/citation.cfm?id=1096143.

[21] John C Mitchell and Krzysztof Apt. *Concepts in programming languages*. Cambridge University Press, 2003, p. 78.

[22] *Oracle Java Reference*. `https://web.archive.org/web/20180927051422/https://docs.oracle.com/javase/7/docs/api/java/util/Random.html`. Accessed: 2018-10-21.

[23] Radek Pelánek. "BEEM: Benchmarks for explicit model checkers". In: *International SPIN Workshop on Model Checking of Software*. Springer. 2007, pp. 263–267.

[24] Sander de Putter, Anton Wijs, and Dan Zhang. "The SLCO Framework for Verified, Model-Driven Construction of Component Software". In: *International Conference on Formal Aspects of Component Software*. Springer. 2018, pp. 288–296.

[25] SMJ San de Putter and AJ Wijs. "A formal verification technique for behavioural model-to-model transformations". In: *Formal Aspects of Computing Formal Aspects of Computing* 30.1 (2018), p. 1.

[26] Bran Selic. "The pragmatics of model-driven development". In: *IEEE software* 20.5 (2003), pp. 19–25.

[27] Jan Smans, Bart Jacobs, and Frank Piessens. "VeriFast for Java: A tutorial". In: *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. Springer, 2013, pp. 407–442.

[28] Dave Steinberg et al. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[29] MF Van Amstel et al. "Transforming process algebra models into UML state machines: Bridging a semantic gap?" In: *International Conference on Theory and Practice of Model Transformations*. Springer. 2008, pp. 61–75.

[30] Jeroen Peter Marie Voeten. *POOSL: An object-oriented specification language for the analysis and design of hardware/software systems*. Eindhoven University of Technology, Faculty of Electrical Engineering, 1995.

[31] John Von Neumann. "First Draft of a Report on the EDVAC". In: *IEEE Annals of the History of Computing* 4 (1993), pp. 27–75.

[32] Anton Wijs and Luc Engelen. "REFINER: towards formal verification of model transformations". In: *NASA Formal Methods Symposium*. Springer. 2014, pp. 258–263.

[33] Dan Zhang. “From concurrent state machines to reliable multi-threaded Java code”. In: *IPA Dissertation Series* 2018 (2018).

List of Listings

Appendices

Appendix A

SLCO Grammar

```
SLCOModel:
  'model' name=NID '{'
  ('actions' actions+=Action)?
  ('classes' classes*=Class)?
  ('objects' objects*=Object[','])?
  ('channels' channels*=Channel)?
  '}'
;

Class:
  name=NID '{'
        ('variables' variables*=Variable)?
        ('ports' ports*=Port)?
        ('state machines'
                statemachines*=StateMachine
        )?
  '}'
;

Object:
  name=NID ':' type=[Class] '(' assignments*=Initialisation[','] ')'
;

Initialisation:
  left=[Variable] ':=' (right=INT|right=BOOL|('['(rights+=INT[',']|rights+=BOOL
       [','])']'))
;

Channel:
  name=NID '(' type*=Type[','] ')'
  ((synctype='async' ('[' size=INT ']')? (losstype='lossless'|losstype='lossy')
  'from' source=[Object]'.'ports=[Port] 'to'
  target=[Object]'.'ports=[Port])
  |
  (synctype='sync' 'between' source=[Object]'.'ports=[Port] 'and'
  target=[Object]'.'ports=[Port])
  )
;

StateMachine:
  name=NID '{'
        ('variables'
                variables*=Variable
        )?
        'initial' initialstate=State
        ('states' states*=State)?
        ('transitions' transitions*=Transition)?
  '}'
```

```
;

State:
  name=NID
;

Port:
  name=NID
;

Transition:
  (priority=INT':')?
  ((source=[State] '->' target=[State])|
  ('from' source=[State] 'to' target=[State]))
  ('{' statements*=Statement[';'] (';')? '}')?
;

Statement:
  (Composite | ReceiveSignal | SendSignal | Delay | Assignment | Expression )
;

Assignment:
  left=VariableRef ':=' right=Expression
;

Composite:
  '[' (guard=Expression ';')? assignments*=Assignment[';'] ']'
;

ReceiveSignal:
  'receive' signal=NID '(' params*=VariableRef[','] ('|' guard=Expression)? ')' '
       from' target=[Port]
;

SendSignal:
  'send' signal=NID '(' params*=Expression[','] ')' 'to' target=[Port]
;

Expression:
  left=ExprPrec4 ((op='or'|op='xor'|op='and'|op='&&'|op='||') right=Expression)?
;

ExprPrec4:
  left=ExprPrec3 ((op='!='|op='='|op='<>'|op='<='|op='>='|op='<'|op='>') right=
       ExprPrec4)?
;

ExprPrec3:
  left=ExprPrec2 ((op='+'|op='-') right=ExprPrec3)?
;

ExprPrec2:
  left=ExprPrec1 ((op='*'|op='/'|op='%') right=ExprPrec2)?
;

ExprPrec1:
  left=Primary (op='**' right=ExprPrec1)?
;

Primary:
  (sign='+'|sign='-'|sign='not')? (value=INT | value=BOOL | '(' body=Expression ')'
       | ref=ExpressionRef)
;

Delay:
  'after' length=INT 'ms'
;

ExpressionRef:
  ref=NID ('['index=Expression']')?
;

Action:
  name=NID
;
```

```
VariableRef:
  var=[Variable] ('['index=Expression']')?
;

Variable:
  (type=Type?) name=NID (':=' (defvalue=INT | defvalue=BOOL|('['(defvalues+=INT
       [',']|defvalues+=BOOL[',']) ']')))?
;

Type:
  (base='Integer' | base='Boolean' | base='Byte') ('[' size=INT ']')?
;

Keyword:
  'actions' | 'model' | 'classes' | 'ports' | 'state machines' | 'variables' | '
       initial' | 'state' | 'transitions' | 'from' | 'to' | 'send' | 'receive' | '
       objects' | 'channels' | 'Boolean' | 'Integer' | 'Byte' | 'async' | 'sync' | '
       lossless' | 'lossy' | 'between' | 'and' | ':=' | 'after' | 'ms' | 'not' | '-'
        | '+' | 'or' | 'xor' | 'and' | '==' | '<>' | '<=' | '>=' | '<' | '>' | 'mod'
        | '*' | '/' | '**' | 'true' | 'false'
;

NID:
  !Keyword ID
;
```