Finding Deadlocks in Open Input-Output Symbolic Transition Systems by Model-Transformation to Promela

Lucas Steehouwer

lucas.steehouwer@student.uva.nl

March 29, 2022, 103 pages

Academic supervisor: Steven de Rooij, s.derooij@uva.nl
Daily supervisor: Theo Ruys, ruys@axini.com
Host organisation: Axini (Axini.com)

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering

Abstract

In the field of Model Based Testing (MBT) Symbolic Transition Systems (STSs) are used to formally specify the behavior of a System Under Test (SUT). Based on this specification, the MBT environment automatically generates, executes, and evaluates test cases. However, the specification itself may contain unwanted behavior. Testing whether the SUT passes all tests then means that unwanted behavior may be marked as permissible. With model checking we can verify that a specification exhibits certain desirable properties, allowing us to verify the specification before we use it for MBT.

Axini is an Amsterdam-based company specializing in MBT and have, to this end, developed the Axini Modeling Language (AML). The behavior of AML models resembles that of a particular class of STS, namely pairwise-synchronizing Input-Output Symbolic Transition Systems (IOSTSs). A particular property of these models is that they may be open, meaning that a model's behavior is dependent on the behavior of an external agent. We have developed a proof-of-concept software solution that is capable of verifying deadlock-freedom of AML models. To achieve this we translate AML models to Promela, the input language of the SPIN model checker.

A substantial part of this thesis details this model-transformation procedure. We have designed and implemented a pipeline that executes all the steps required to deliver a Promela program. If SPIN detects an erroneous state, then it reports the steps to get to this state. Our pipeline produces meta-data that we use to present these reports using symbols from the original AML model.

We have tested our prototype and found that, while our translation did introduce superfluous states, we did detect all errors in the models we tested. Thus, we have created a working prototype that shows that such a translation is indeed possible. In addition, the translation we define can be generalized to also work for pairwise-synchronizing open IOSTSs. Because of our use of an existing model checker, SPIN, we believe that our prototype can be extended to verify other safety properties with relative ease. However, some of the obstacles that we had to overcome suggest that either extending SPIN and Promela, or building a new model checker entirely, might be a better approach to verifying properties of AML models and related formalisms.

Contents

Chapter 1

Introduction

Software testing is an important step of any software development process. By testing software, one gains confidence in the software doing what it is supposed to do; it conforms to a certain specification. Many software testing techniques exist, one of which is Model Based Testing (MBT). With MBT, an implementation, called the System Under Test (SUT), is considered a black-box. This box exhibits observable behavior, and this behavior must conform to a specification. Specifications come in many forms with varying degrees of formality. However, for MBT the specification *must* be formal; there may be no confusion about what behavior the specification entails. Based on the specification, test cases are generated, executed, and evaluated automatically. The collection of test cases is called the test suite. Typically, the suite is required to be sound; no conforming implementation will fail the suite. In other words, the testing procedure based on this suite may not give false negatives [1]. Figure 1.1 provides an overview of MBT. If the specification itself contains unwanted behavior, then testing whether the SUT conforms to it is a waste of effort. Unknowingly, we have a specification we do not want, and by performing tests we gain confidence in the SUT conforming to it, leading us to believe that the SUT exhibits the correct behavior, while in reality we have found evidence that suggests the SUT conforms to an undesired specification, and undesired behavior may therefore be marked as permissible.

A solution to this problem is to model check the specifications themselves. Model checking is the act of verifying whether a system satisfies certain desirable properties or not [3]. The word "system" may refer to both implementations and specifications. By model checking the specification we verify that it does not contain undesired behavior, and we can then proceed to use it to test the SUT with MBT.

Axini is an Amsterdam-based company specializing in MBT and have to this end, developed the Axini Modeling Language (AML). AML is a Domain Specific Language (DSL) used to specify the desired behavior of a SUT. AML *models* are structurally and semantically close to Input-Output Symbolic Transition Systems (IOSTSs) [4], which are special kinds of Symbolic Transition Systems (STSs) [5]. STSs and IOSTSs are formalisms specifically invented for MBT [4]–[6]. A model checker capable of verifying properties of IOSTSs or AML models does not exist yet, and there is little available research on this topic. The goal of this thesis is to

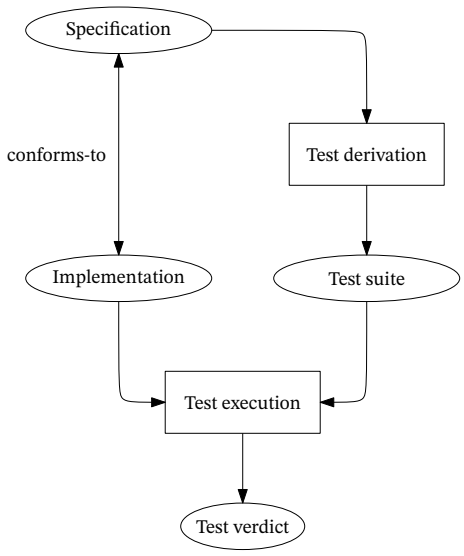

Figure 1.1 – An overview of MBT. Reproduced from [2].

verify *safety* properties [7] of AML models, using a model checker. Thus, we provide an initial step towards the formal verification of properties of AML models and related formalisms.

We can now state the main research question of this thesis.

Research Question
How can we verify safety properties of AML models?

The construction of a model checker capable of verifying safety properties is no easy task. Instead, we make use of an off-the-shelf model checker, SPIN [8]. We motivate this decision in more detail in the next section. To answer the research question we will perform a model-transformation; we will translate AML models to Promela, the input language of SPIN. To facilitate this transformation, we have designed and implemented a pipeline. The pipeline starts with an AML model, prepares it, translates it to Promela, and verifies it with SPIN. If an erroneous state is detected, then the steps to traverse to this state are shown to the user.

In the next section we will provide a high-level overview of the problems we need to solve to make the pipeline work and answer our research question. Each of these problems are then discussed in more detail in subsequent chapters. We then verify the results produced by our pipeline by comparing them with other known solutions. While AML models are semantically close to IOSTSs, they are not the same. We dedicate a chapter to explaining the differences between the two, and how this affects the generalizability of our solution.

1.1 Problem Description

We want to verify safety properties of AML models. The envisioned use case is that a user creates an AML model specifying the behavior of a particular SUT such that it can be tested using MBT techniques. Before performing these tests, the user verifies that the AML model

satisfies a number of desirable properties. Of course, this implies that the user must specify these properties first. If an erroneous state is found, then the user is presented a sequence of steps that describe how the undesired state is reached. The user can then make the necessary changes to the model and try again.

Building a model checker is no trivial task. Not only does it require substantial effort to implement the desired functionality, verifying whether the model checker does not produce false results is another challenge. Instead of developing a completely new model checker from scratch, we made the decision to use an existing one, SPIN. SPIN is a state-of-the-art model checker capable of verifying properties of large systems. Its input language, Promela, is simple and contains all the building blocks necessary to specify complex systems. The kinds of properties SPIN can verify range from simple ones, like assertions and deadlock-freedom, to complicated LTL [9] formulas. The decision to use an existing model checker has had far-reaching consequences on the rest of this thesis. We wanted to leverage SPIN's extensive model checking capabilities, allowing us to verify many kinds of safety properties. However, translating AML models to semantically equivalent Promela programs turned out to be harder than we initially thought. Consequently, we have reduced the scope of this thesis. The prototype we have delivered with this thesis is capable of verifying deadlock-freedom of AML models. We expect that adding support for the verification of other properties is relatively easy due to our decision to use SPIN. We reflect on our decision for SPIN in more detail in Chapter 7.

An AML model consists of one or more processes that communicate with one another by sending parameterized messages over one or more *channels*. A message consists of a *label*, accompanied by *label parameters*. The state of a process is determined by the values for its local variables, called *process variables*, and its *location*[1]. Each process contains a set of transitions that describe how the process may change state. The execution of a transition may be constrained by a logical expression. There are two kinds of transitions: unobservable transitions, also called τ-steps, and message exchanges. The constraints of τ-steps may refer only to the process variables, whereas the constraints of message exchanges may refer to both the process variables and the message's label parameters. When a process executes a τ-step it changes its own state. In contrast, when a process exchanges a message with another process, both processes change state concurrently. Say we have two processes, a sender and a receiver, and a particular message. The processes may exchange this message if and only if both are in a state where they may respectively send and receive it. Once the message has been exchanged, both processes update their state. The communication of the message and the subsequent state update of both processes occur in a single, non-divisible step. This communication mechanism is known as *handshaking*, or *synchronization*. If either process is not ready, the handshake may not occur; neither process changes state.

The models we verify may be *open*. An open model contains behavior that is dependent on the input and output of a particular external agent, the *environment*. Open AML models communicate with the environment over *external* channels via handshaking. SPIN is, like most model checkers, not designed to verify properties of open systems. It can only verify

[1] The term "location" is, strictly speaking, an STS-related term, not an AML-related term. Perhaps confusingly, the "state" keyword in an AML model denotes a location.

properties of *closed* systems. Thus, before we can verify an open model, we must first transform it to a closed one. This is known as *closing* a model. A common approach to closing an open system is to create an additional process that simulates the environment, and add this process to the original system, replacing the environment [10]. This additional process is called the *environment process* or *environment model*. We adopt this approach as well. We add the environment process to the original model, and all communication with the environment is replaced by communication with environment process. The entire closure procedure is explained in more detail in Chapter 3.

The translation from a closed AML model to a semantically equivalent Promela program is non-trivial. There are three main obstacles we need to overcome:

1. Promela and AML support different types. Say we have an AML model containing one or more variables of a type not supported by Promela. We must come up with a way to translate these variables to valid Promela.
2. Whether two processes may exchange a particular message depends on the local state of both processes, and the message parameters. When they synchronize, both processes update their state. Both updates occur in a single atomic step. We must replicate this synchronization mechanism in Promela.
3. In AML, message parameters are implicit. An AML transition stating that a message should be either sent or received may define a constraint that partially determines whether the transition is enabled. Whether this constraint holds depends on the values for the process's local variables, and the message parameters. Together, the constraints of the sender and the receiver determine the set of parameter mappings that will make both transitions enabled given the values for the local variables of both processes. However, in Promela message parameters are sent explicitly. Thus, we must extract these parameter mappings, and transcribe them into the Promela program.

If SPIN finds an error it produces a report showing how the erroneous state is reached. This report consists of steps of the Promela program. In order for these steps to make sense to the user, we will need to translate these Promela steps back to steps in the AML model. The user can then determine how a deadlock in their model is reached.

We provide a more detailed description of the pipeline in Chapter 2. Each problem we have described will be further described in subsequent chapters.

1.2 Scope

We have decided to limit the kinds of AML models our prototype can successfully verify. By limiting our scope to these kinds of models, we were able to simplify parts of the pipeline. This freed up resources to spend on other parts we consider more important in regard to answering our research question. If the reader is not familiar with AML, we recommend reading the overview of the language found in Appendix A.1 before proceeding.

The first limitation pertains to the data types we allow in AML models. The supported types of AML and Promela do not fully overlap. We only support a small set of AML types:

integers, booleans, and strings. A model containing variables of a different type is not in scope. Promela has support for integers and booleans, but not for strings. In Section 4.5 we show how we can add support for additional data types in SPIN, including strings.

The second and third limitations pertain to the allowed constraint and update expressions, respectively. In Section 4.4 we explain the reason for these restrictions, and we point to known techniques that can be used to lift them. An update expression consists of zero or more assignments; they assign new values to the process variables. The restriction we impose on these assignments is that the right-hand side may not contain function calls. In addition, if the variable on the left-hand side is of type string, then the right-hand side of the assignment must be either a string literal, or a reference to another variable, also of type string.

The kinds of constraints we allow for stimuli differ from those we allow for responses. For responses, we only allow constraints that are disjunctions of conjunctions of equalities between atoms. We define atoms here as either literal values, or references to other variables. In every conjunction every label parameter must be equated to an atom. Process variables may be equated to other literals, label parameters, and other process variables. Label parameters, however, may only be equated to literals and process variables; they may not be equated to other label parameters. For stimuli, we again require that the constraint is a disjunction of conjunctions. In each conjunction each label parameter *must* be equated to a literal value. They may additionally be equated to a process variable in the same conjunction. Sub-expressions constructed over the process variables of types integer and boolean are allowed in each conjunction. For example, if $\square$ is a process variable, the sub-expression $\square < 10$ is allowed. If $\square$ is a label parameter, then this sub-expression is not allowed. For every stimulus label on an internal channel there exists a corresponding response label. For every parameterized label, we require that for every stimulus/response pair with this label, at least one of the transitions has a non-empty constraint. Constraints on external transitions must be non-empty.

1.3 Related Work

Model checking was invented in the early 1980s with the works of [11], [12]. These works showed how to algorithmically prove that a system is a model of a particular property [13]. The promise of model checking was simple, but powerful: given a system and a particular property, one can verify whether the system exhibits this property or not at the push of a button. If the verification process fails the model checker produces a counterexample, showing exactly how to arrive at an undesired state. These properties are often specified in temporal logic, like CTL [12], [14], and LTL [9]. Logics for specifying properties and the mechanisms to verify them are still active areas of research [15]–[20].

From day one the model checking world has had to deal with the *state space explosion* problem [21]. This is the problem where a system contains so many states that verifying whether a certain property holds becomes infeasible in terms of time, memory, or both. Over the years many methods have been invented to combat this problem, some of which have found their way into model checking tools. Two prominent examples of such methods are *partial order reduction* [22]–[26], and *predicate abstraction* [27], [28]. Under certain

conditions, it is even possible to model check systems with an infinitely large state space [29], [30]. Most model checkers do not rely on a single method to combat the state space explosion problem. This is for a good reason. As research from [31] shows, there is no *silver bullet*: no single method will completely alleviate the state space explosion problem. A more fruitful approach is to implement multiple simple methods, rather than implementing a single complicated method.

An open system is a reactive one [32]. Its behavior is influenced by communication with an outside agent, the environment. Most model checkers are only designed to verify properties of closed models. The only model checker we know capable of verifying properties of open models is Mocha [20]. Unfortunately, Mocha seems to be no longer maintained, and research surrounding it lacks detail regarding the specific challenges of verifying open systems. Most research regarding the verification of open models has focused on closing models without incurring too large of a cost in terms of state space size increase. A common approach is to add an environment process, and to redirect all communication with the environment to this new process. To avoid a state space blow-up, either the environment process is abstracted [33]–[37], or the communication mechanism is [10]. For this project we also integrate an environment process into the input AML model, replacing the environment. However, we do not abstract the environment in any way.

Parallel in time to the model checking field, the field of MBT found its inception. Originally, the theories for testing the conformance of a SUT to its specification were based on Labeled Transition Systems (LTSs) [1]. However, the state space explosion problem also reared its ugly head here. To alleviate this problem, STSs were invented together with new test theories [4], [5], [38]. In essence, an STS is an augmented LTS allowing for data-dependent behavior. An STS can be unfolded into an LTS; for any STS there is an equivalent LTS. However, due the structure of an STS, more efficient testing theories can be applied.

A model checker capable of verifying properties of STSs does not exist. A temporal logic for specifying these properties does exist [39], but this has not yet found its way into an available tool. We have found two works related to model checking STSs, [40], and [41]. The study of [40] attempt to check whether a particular state of an AML model is reachable using an SMT-solver. The work of [41] is the main inspiration of this thesis. In this work AML models are closed and transformed into the input language of the DiVinE model checker [42], DVE. The LTSMin model checker [43] carries out the verification. This thesis follows the same approach: we transform AML models to Promela programs after closing them first. There are also important differences. First, we show how AML data types not supported by SPIN and Promela can still be used. Second, we apply a different method of closing models. In [41] an environment process is generated for each process in the AML model, but we generate just a single one. Third, [41] does not explain the translation process in detail. We spend a substantial amount of effort explaining the translation from AML to Promela. Fourth, [41] notes how particular deadlocks in the AML model occur as livelocks in the DVE model. A large part of the thesis is dedicated to preventing this, resulting in a Promela program that is semantically closer to the AML model than the DVE programs produced by [41].

Chapter 2

The Model Checking Pipeline

In this chapter we describe the model checking pipeline we sketched in the previous chapter in more detail. The goal of the pipeline is to verify safety properties of AML models. A major part of the pipeline involves the translation from AML to Promela. For readers not familiar with either language we have included short overviews of both languages in the Appendix. The overview for AML is found in Appendix A.1, and the overview for Promela is found in Appendix A.2.

Figure 2.1 shows the overall pipeline of the application. We start with a model provided in AML syntax. The goal of the pipeline is to translate this model into a semantically equivalent model in Promela syntax, and then feed this into the SPIN model checker. If SPIN finds an error we must undo the translations. This way, the trace SPIN produces shows how an undesired state is reached in terms of steps in the AML model, rather than steps of the Promela code.

We first use Axini's libraries to parse the model. The parsed result is the same model in data structures from Axini. We then close the model; we generate an environment process and an internal channel for communicating with it. We add the environment to the original model. Additionally, all external channels are removed, and all labels assigned to external channels are re-assigned to the environment channel. We describe the closure procedure in more detail in Chapter 3.

The SPIN model checker takes models provided in the Promela language as input. We therefore translate the closed model to Promela, and save the resulting code. This is then fed into SPIN, which generates C source text describing a verifier for this model. After compilation, we have a verifier specific to our model. If this verifier detects an error it will produce a trail in a separate file. In Chapter 4 we describe our mapping from closed AML models to Promela.

Both the closure and Promela conversion processes produce meta information describing how their respective inputs were transformed. We have created a reversal mechanism that uses this information to translate a trace back to constructs used in the original AML model. We describe this reversal mechanism in more detail in Section 2.3. If the AML model contains

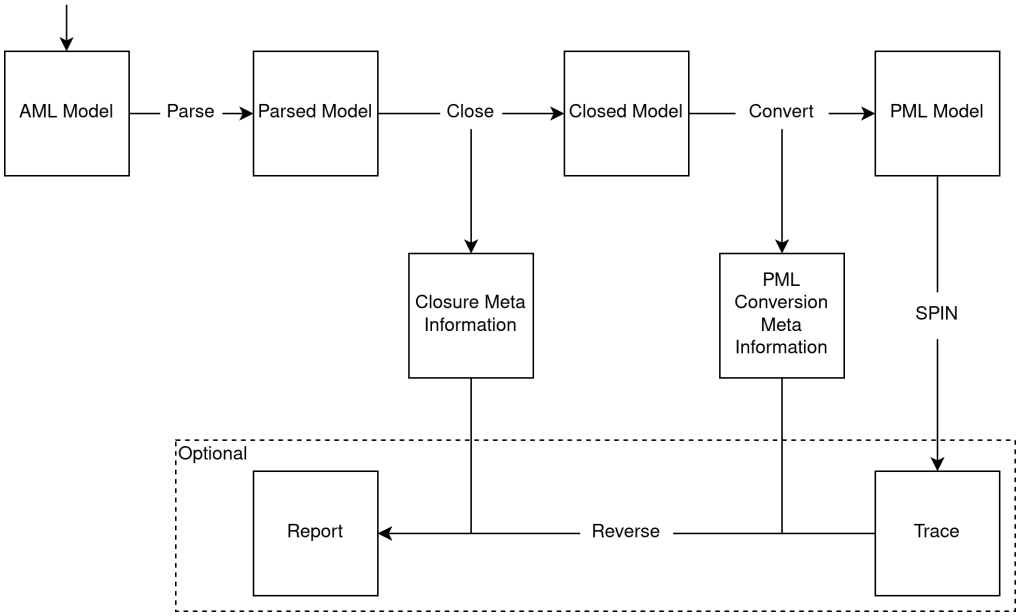

Figure 2.1 – The overall pipeline of the software implementation. The optional segment is only executed when SPIN detects an error.

a deadlock, then the result of the pipeline is a report describing the steps to the deadlocked state. If the model is deadlock-free, then the report is empty.

2.1 Running Example

To illustrate the steps of the pipeline we have created an example AML model, shown in Listing 2.1. This model contains all edge cases that could make the closure and Promela translation problematic; it contains multiple internal channels, multiple labels with the same name, a □-step, and at least one deadlock. This is a toy model. In no way do we intend for this model to exhibit useful behavior. Its sole purpose is to act as an illustration.

2.2 Implementation

Axini has developed a library used for parsing AML models, and performing operations on them. This library is called the `sts-engine`. The prototype has been developed as a new part of this engine. It is, therefore, embedded in the `sts-engine` code. Both AML and the `sts-engine` are implemented in Ruby[1]. As a consequence, the prototype is implemented in Ruby as well.

The implementation closely follows the schematic of the pipeline. A `ModelChecker` object wraps around the pipeline and takes care of the plumbing; as part of its initialization it receives a parsed AML model. It feeds this model into a function that yields a `Closure` object. This object contains a `ClosedModel`, which `ModelChecker` passes on to a `PromelaConverter` object, yielding a `PromelaConversion` object. The `ModelChecker` then feeds this into a

[1] `https://www.ruby-lang.org`

```
internal 'int1', 'int2'
external 'ext'
timeout 1

process('A') {
  channel('int1') {
    stimulus 'a'
    response 'k', { 'x' => :integer }
  }
  channel('int2') {
    stimulus 'a'
  }
  channel('ext') {
    stimulus 'set', { 'x' => :integer }
    response 'success', { 'new_i' => :integer }
  }
  var 'i', :integer, 0
  var 'reached_home_by', :string, "init"

  state 'home'
  receive 'a', on: 'int1'
  goto 'waiting'

  state 'waiting'
  choice {
    o { receive 'set', constraint: '(x == 1 || x == 2) && i == 1', update: 'i =
        □  x + 1'
        send 'success', constraint: 'new_i == i'
        goto 'set' }
    o { send 'k', constraint: 'x == i', update: 'i = i + 1; reached_home_by = "
        □ k"'
        goto 'home' }
  }
  state 'set'
  choice {
    o { receive 'a', on: 'int2', update: 'i = 1; reached_home_by = "a"'
        goto 'home' }
    o { tau constraint: 'i == 0', update: 'reached_home_by = "tau"'
        goto 'home' }
  }
}

process('B') {
  channel('int1') {
    response 'a'
    stimulus 'k', { 'x' => :integer }
  }
  channel('int2') {
    response 'a'
  }
  channel('ext') {
    stimulus 'set', { 'x' => :integer }
  }
  var 'i', :integer, 0

  state 'home'
  send 'a', on: 'int1'
  goto 'comm'

  state 'comm'
  choice {
    o { receive 'k', constraint: 'x == i && i < 3', update: 'i = i + 1'
        goto 'home' }
    o { send 'a', on: 'int2'
        goto 'home' }
    o { receive 'set', constraint: '(x == 0 && i == 0) || (x == 1 && i == 0)'
        goto 'comm' }
  }
}
```

Listing 2.1 – The AML example model we use to illustrate all major steps of the pipeline.

`PromelaPrinter` object, and tells it to print its result, Promela code, to a specific file. The `ModelChecker` then feeds this file into SPIN, which generates code for a verifier for the Promela model. The `ModelChecker` uses the GNU Compiler Collection (`gcc`)[2] to compile the verifier. After successful compilation, it runs the verifier. Once the verifier has completed its verification, the `ModelChecker` checks whether a trail file has been produced. If so, it will use the `Closure` and `PromelaConversion` objects to translate the trail back to steps of the AML input model. The `ModelChecker` then has a report to show to the user, and prints this to `stdout`.

Figure 2.2 shows the UML class diagram for the prototype. Apart from the classes in the `Axini` package, all classes are contained in the `Experiments` module. For convenience, we have left these module descriptions out of the diagram. Note that the `ModelChecker` is instantiated with a parsed AML model as one of its arguments. We have created a class called `Models` that is used as a helper for retrieving and parsing AML models. Furthermore, the `Models` class contains a `verify_model` method, which, given a model name as a string, retrieves and parses the model, and verifies it using a `ModelChecker` object.

Remember that the envisioned use case involves a user creating an AML model, and signaling the modeling environment to verify their model once ready. As such, we have not created a standalone script that checks models. Instead, to verify a model one must open a ruby session for the `sts-engine`, and call the method for verifying models:

```
cd sts-engine
bundle console
> Experiments::Models.verify_model "my_model"
```

The name “my_model” can be substituted for any existing model name. All models are placed in a directory of the testing environment. The prototype contains a simple hard-link to this directory. As an aid, the user may request the names of all known models:

```
cd sts-engine
bundle console
> Experiments::Models.list_models
```

2.3 Reporting Errors

The closure and Promela translation step are intricate enough to warrant their own chapter. In contrast, the reversal mechanism is quite straightforward. For this reason we do not dedicate an entire chapter to this topic. We recommend the reader to first read Chapter 3 and Chapter 4 before reading this section.

In addition to verifying whether a system exhibits a certain property, a model checker provides counter-examples when a property is violated. If the AML input model contains a deadlock, then this must be reported to the user. SPIN will provide a sequence of steps that lead to an undesired state. However, these steps are steps of the Promela program, not the original AML model. We must translate these sequences such that they make sense to the user; they must be sequences of steps of the original AML model.

[2] `https://gcc.gnu.org`

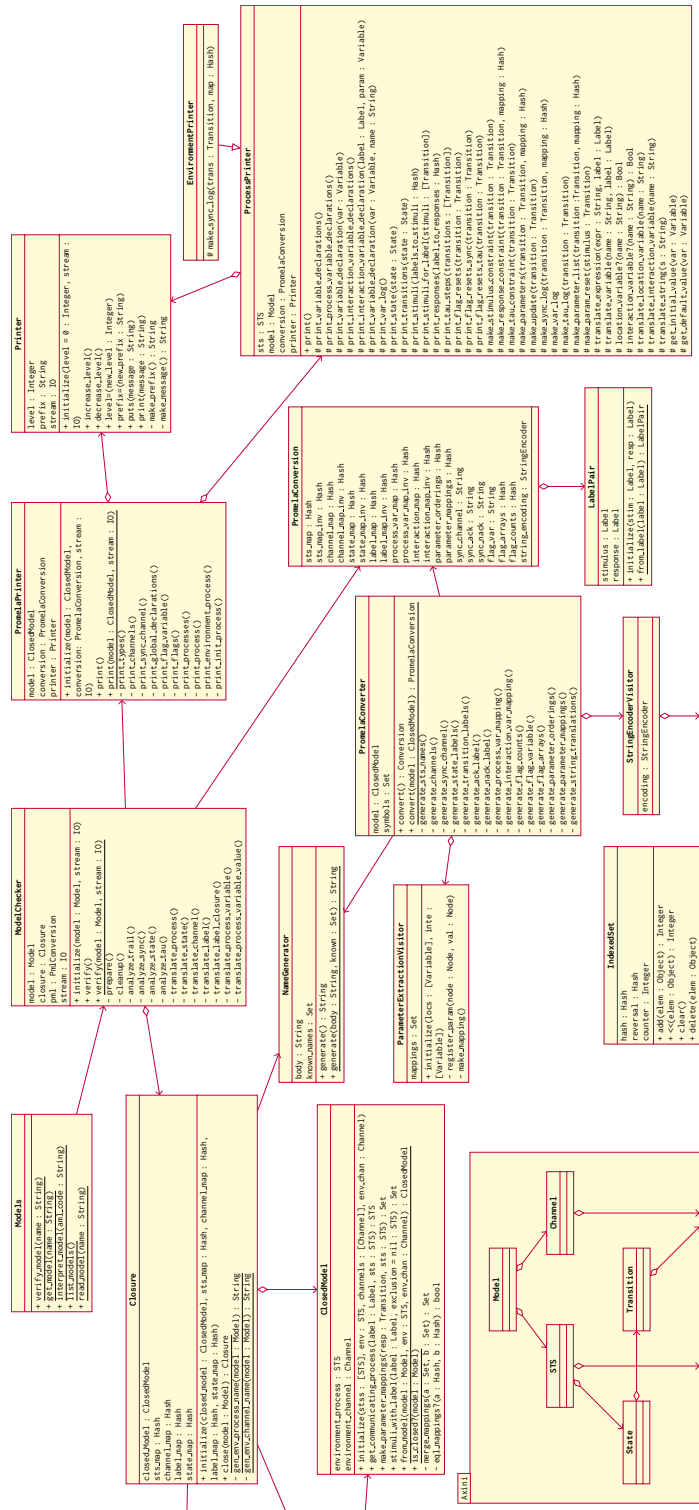

Figure 2.2 – UML Class diagram showing the classes and their interaction of the prototype.

```
 13:  proc  2 (B) complete.pml:152 (state 30) [sync?ACK]
B[label_B_home]: chan_a ! a -> label_B_comm
 14:  proc  2 (B) complete.pml:153 (state 26) [D_STEP153]
 15:  proc  2 (B) complete.pml:159 (state 25) [goto label_B_comm]
 16:  proc  1 (A) complete.pml:40 (state 45)  [goto label_A_waiting]
 17:  proc  1 (A) complete.pml:48 (state 162) [chan_k[(( (( !(flags_k[0])&&(i==i
     □ ))) ? (0) : (1) )]!k,i]
 18:  proc  2 (B) complete.pml:167 (state 154)  [chan_k[0]?k,k_x_0]
 19:  proc  2 (B) complete.pml:168 (state 68) [(((k_x_0==i_0)&&(i_0<3)))]
B[label_B_comm]: chan_k ? k(0) -> label_B_home
B: [i_0 = 1]
 20:  proc  2 (B) complete.pml:169 (state 63) [D_STEP169]
 21:  proc  2 (B) complete.pml:178 (state 61) [sync!ACK]
 22:  proc  1 (A) complete.pml:68 (state 159) [sync?ACK]
A[label_A_waiting]: chan_k ! k(0) -> label_A_home
A: [i = 1, reached_home_by = 1]
 23:  proc  1 (A) complete.pml:69 (state 155) [D_STEP69]
 24:  proc  1 (A) complete.pml:79 (state 154) [goto label_A_home]
 25:  proc  2 (B) complete.pml:179 (state 62) [goto label_B_home]
 26:  proc  2 (B) complete.pml:151 (state 33) [chan_a[(( (( !(flags_a[0])&&1)) ?
     □  (0) : (1) )]!a]
 27:  proc  1 (A) complete.pml:28 (state 54)  [chan_a[0]?a]
 28:  proc  1 (A) complete.pml:29 (state 51)  [(1)]
A[label_A_home]: chan_a ? a -> label_A_waiting
 29:  proc  1 (A) complete.pml:30 (state 46)  [D_STEP30]
 30:  proc  1 (A) complete.pml:39 (state 44)  [sync!ACK]
 31:  proc  2 (B) complete.pml:152 (state 30) [sync?ACK]
B[label_B_home]: chan_a ! a -> label_B_comm
 32:  proc  2 (B) complete.pml:153 (state 26) [D_STEP153]
 33:  proc  2 (B) complete.pml:159 (state 25) [goto label_B_comm]
 34:  proc  1 (A) complete.pml:40 (state 45)  [goto label_A_waiting]
```

Listing 2.2 – Partial trace of the example model.

When the verifier for a Promela program is run and detects an error, it produces a trail file. The verifier can then use the information in this file to produce a counter-example. The verifier supports a large set of compile and runtime options that influence its behavior. Assume we have written the generated Promela program to a file called `model.pml`. We compile and run the verifier with the following options:

```
spin -a model.pml
gcc -DMEMLIM=6144 -DXUSAFE -DNOBOUNDCHECK -DNOFAIR -DSAFETY -DREACH -O3 -w -o
    □ pan pan.c
./pan -i -m50000000
```

On the first line we instruct SPIN to generate code for a verifier for the provided model. We compile this code on the second line. We use a set of compile time options. All options prefixed with `-D` are options that influence the behavior of the verifier. With the `-DMEMLIM` option we impose a memory usage limit. We prohibit the verifier from using more than 6144 megabytes of memory. The `-DXUSAFE`, `-DNOBOUNDCHECK`, `-DNOFAIR`, and `-DSAFETY` options optimize the verifier for our use case. Each of these four options disables a certain kind of validity check that we do not require for deadlock detection. We compile the verifier with a high level of optimization, `-O3`. This results in longer compile times, but it does make the verifier faster. This optimization pays off for large models. With the `-w` option we turn off the reporting of compilation warnings. On the last line we run the model checker. Further, we impose a very large depth limit using the `-m` option. This depth limit is much larger than needed for most models. But, for some models such a large depth is needed. There may be multiple paths to an erroneous state, and the path that SPIN finds may not be the shortest

```
1  B: [i_0 = 0]
2  A: [i = 0, reached_home_by = 4]
3  A[label_A_home]: chan_a ? a -> label_A_waiting
4  B[label_B_home]: chan_a ! a -> label_B_comm
5  B[label_B_comm]: chan_k ? k(0) -> label_B_home
6  B: [i_0 = 1]
7  A[label_A_waiting]: chan_k ! k(0) -> label_A_home
8  A: [i = 1, reached_home_by = 1]
9  A[label_A_home]: chan_a ? a -> label_A_waiting
10 B[label_B_home]: chan_a ! a -> label_B_comm
11 B[label_B_comm]: chan_k ? k(1) -> label_B_home
12 B: [i_0 = 2]
13 A[label_A_waiting]: chan_k ! k(1) -> label_A_home
14 A: [i = 2, reached_home_by = 1]
15 A[label_A_home]: chan_a ? a -> label_A_waiting
16 B[label_B_home]: chan_a ! a -> label_B_comm
17 B[label_B_comm]: chan_k ? k(2) -> label_B_home
18 B: [i_0 = 3]
19 A[label_A_waiting]: chan_k ! k(2) -> label_A_home
20 A: [i = 3, reached_home_by = 1]
21 A[label_A_home]: chan_a ? a -> label_A_waiting
22 B[label_B_home]: chan_a ! a -> label_B_comm
```

Listing 2.3 – The full trail for the example model, showing only the evaluations of the `printf` statements.

one. To reduce the cognitive load on the user, we want to find the shortest path. The `-i` option instructs the model checker to find the shortest path to each erroneous state it finds. This functionality is only supported when the verifier is compiled with the `-DREACH` option.

Consider Listing 2.2. It shows part of the trail produced for the example model. The complete trail consists of 100 steps. We only show part of the trail for brevity reasons. While this trail is human-readable, it contains a lot of information pertaining to the Promela translation of the AML model, which is not of interest to the user.

At the end of Section 4.2 we show how we embed `printf` statements in the Promela model. We embed these statements at the appropriate places. For example, knowing that a particular synchronization attempt failed is not relevant to the user. Therefore, this should not be logged. In contrast, a successful synchronization attempt is relevant because it shows how a state on the path to the deadlock state is reached. Thus, such an event should be logged. We can see the results of the evaluation of these `printf` statements on lines 18, 25, 26, 30, 31, 38, and 42. By running the generated model checker with the `-S` option, only these statements are printed. All other statements are filtered out.

Listing 2.3 shows the trail when running the generated model checker with the `-S` option. This is the entire trail. It is much shorter as no superfluous information is logged. There are two kinds of log statements. An example of the first kind is shown on the first line. This line shows the current values for the variables of process `B`. An example of the second kind is shown on line 3, where process `A` reports receiving an `a` label. The first kind is technically not necessary to determine how the deadlock is reached. However, requiring the user to manually track the values of the local variables of each process may be a tall order. By including these statements we make the process easier, reducing the cognitive load.

Both these trails refer to elements of the Promela program. Thus, in order to make sense of

```
B: [i = 0]
A: [i = 0, reached_home_by = "init"]
A[home]: int1 ? a -> waiting
B[home]: int1 ! a -> comm
B[comm]: int1 ? k(0) -> home
B: [i = 1]
A[waiting]: int1 ! k(0) -> home
A: [i = 1, reached_home_by = "k"]
A[home]: int1 ? a -> waiting
B[home]: int1 ! a -> comm
B[comm]: int1 ? k(1) -> home
B: [i = 2]
A[waiting]: int1 ! k(1) -> home
A: [i = 2, reached_home_by = "k"]
A[home]: int1 ? a -> waiting
B[home]: int1 ! a -> comm
B[comm]: int1 ? k(2) -> home
B: [i = 3]
A[waiting]: int1 ! k(2) -> home
A: [i = 3, reached_home_by = "k"]
A[home]: int1 ? a -> waiting
B[home]: int1 ! a -> comm
```

Listing 2.4 – The full translated trail for the example model, showing how a deadlock is reached by following a sequences of steps of the original AML model.

this trail the user would need to inspect the Promela code first. This is not what we want. As shown in Figure 2.1 both the closure and the Promela translation steps produce extra information detailing their transformations. We can use this information to translate the Promela steps back to steps of the original input AML model. This translation step is the last step of the pipeline. Its result is a report showing how an undesired state is reachable by following steps of the original AML model. Listing 2.4 shows the translated report that is presented to the user.

Chapter 3

Closure

The first important step of the pipeline is closing the AML input model. In this chapter we describe what this step entails.

AML supports two kinds of channels; internal channels, and external channels. The closure procedure internalizes all external communication, and removes all external channels. If the model contains at least one process that communicates over an external channel, then it is open. As we briefly described in Section 1.1, we close an open model by creating an *environment process* and replacing all external communication with communication with the environment process. We start by creating an internal channel used for communication between the environment process and the original processes. We call this channel the *environment channel*. Then we create the environment process. This process has only a single state, which we refer to as the home state. For every label on an external channel, we add a transition with the same label to the environment process, but in the opposite direction: a stimulus becomes a response, and a response becomes a stimulus. This transition has the home state as both its source and target state. Thus, each transition of the environment process is a self-loop. Its constraint is simply □. The environment process does not have any location variables that it can update. Therefore, the transition's update expression is empty. Next, we re-assign each external label to the environment channel. This way, all communication on external channels now occurs on the environment channel.

Because AML processes synchronize in pairs, making the external labels internal by re-assigning them to the environment channel gives rise to some issues. It may occur that two or more processes in the original model share at least one external label. That is, there are at least two processes that define a label with the same name and parameter list on an external channel[1]. In AML, it is considered correct to declare two external labels with the same name and parameter list, but a label on an internal channel may only occur in two processes that communicate over this channel. In one process this label is a stimulus, and in the other it is a response. There may not be another process that defines a label with the same name and parameter list on the same channel. This is a deliberate choice regarding the semantics of AML. Consider the following situation involving three processes, `A`, `B`, and `C`: if process `A`

[1] These labels may be assigned to different external channels

wants to send a label x, but both processes B and C can receive it, which process should A send it to? What might happen, is that two external labels with the same name and parameter list, defined in two different processes, are both re-assigned to the environment channel. The result is an incorrect model, because this now internal label is defined in three places: these two processes and the environment process. When we re-assign the external labels to the new environment channel, we must transform the label names such that they are unique: an external label x defined by process A must have a different name from external label x defined by process B. As a result, when the environment process sends a label to or receives a label from another process, it is unambiguous which process it should synchronize with. This way, we preserve the pairwise synchronization semantics of AML.

The model found in Listing 3.1 is the closed version of the example model introduced in Section 2.1[2]. The example model contains one external channel, ext. Both process A and B state that they can receive a set stimulus on this channel. Process A can also send a success response. In the closed model below we see that the external channel has been removed, and an internal channel has been added, internal_env. This is the environment channel. In process A and B we find that the external labels are re-assigned to the internal_env channel. These labels have been renamed such that they are unique; the set label for process A is now called set_A, while the set label for process B is now called set_B. We also find a new process, closure_env. This is the environment process. It has a single state, home, and it can send set responses, and receive success stimuli. Every transition is a self-loop. Notice that none of the transitions of the environment process have constraints or updates defined on them. Whether a process may communicate with the environment process depends entirely on that process itself. For example, the environment process may at any point try to send a set_A message, but this synchronization attempt may only succeed when the value of parameter x is either 1 or 2, and the value of process variable i is equal to 1. Note that the gen_2 state was an anonymous state in original AML model. It has been made explicit by the parsing process.

The method we presented here is just *one* way to close a model. There are others. Any process that is capable of sending all relevant inputs while receiving all outputs on previously external channels is a valid environment process. In fact, the environment may be replaced by *multiple* environment processes, like in the work of [41]. A valid alternative method then, would be to generate an environment process and channel for each process of the input model. An advantage of this approach is that labels no longer need to be renamed. We further reflect on our choice of closure procedure in Chapter 7.

```
1 internal 'int1', 'int2', 'internal_env'
2 timeout 1.0
3 process('A') {
4   channel('int1') {
5     stimulus 'a'
6     response 'k', { 'x' => :integer }
7   }
8   channel('internal_env') {
```

[2] The presented code is not actually produced at any point in the pipeline. After the parsing step, we have a data object representing the input model. The closure algorithm is executed on this object, yielding a data object representing the closed AML model. This code merely acts as an illustration of the closure procedure. In addition, the model is incorrect with respect to the constraints of the responses of the environment process. These constraints may not be empty.

```
    stimulus 'set_A', { 'x' => :integer }
    response 'success_A', { 'new_i' => :integer }
  }
  channel('int2') {
    stimulus 'a'
  }
  var 'i', :integer, 0
  var 'reached_home_by', :string, "init"

  state 'home'
  receive 'a', on: 'int1'
  goto 'waiting'

  state 'waiting'
  choice {
    o { receive 'set_A', constraint: '(x == 1 || x == 2) && i == 1', update: 'i
        □  = x + 1'
      goto 'gen_2' }
    o { send 'k', constraint: 'x == i', update: 'i = i + 1; reached_home_by = "
        □ k"'
      goto 'home' }
  }

  state 'gen_2'
  send 'success_A', constraint: 'new_i == i'
  goto 'set'

  state 'set'
  choice {
    o { receive 'a', on: 'int2', update: 'i = 1; reached_home_by = "a"'
      goto 'home' }
    o { tau constraint: 'i == 0', update: 'reached_home_by = "tau"'
      goto 'home' }
  }
}

process('B') {
  channel('int1') {
    response 'a'
    stimulus 'k', { 'x' => :integer }
  }
  channel('int2') {
    response 'a'
  }
  channel('internal_env') {
    stimulus 'set_B', { 'x' => :integer }
  }
  var 'i', :integer, 0

  state 'home'
  send 'a', on: 'int1'
  goto 'comm'

  state 'comm'
  choice {
    o { receive 'k', constraint: 'x == i && i < 3', update: 'i = i + 1'
      goto 'home'}
    o { send 'a', on: 'int2'
      goto 'home' }
    o { receive 'set_B', constraint: '(x == 0 && i == 0) || (x == 1 && i == 0)'
      goto 'comm' }
  }
}

process('closure_env') {
  channel('internal_env') {
    response 'set_A', { 'x' => :integer }
    stimulus 'success_A', { 'new_i' => :integer }
    response 'set_B', { 'x' => :integer }
```

```
76    }
77
78    state 'home'
79    choice {
80      o { send 'set_A'
81        goto 'home' }
82      o { receive 'success_A'
83        goto 'home' }
84      o { send 'set_B'
85        goto 'home' }
86    }
87  }
```

Listing 3.1 – The closed variant of the example model produced by our pipeline.

Chapter 4

Promela Conversion

A major part of the pipeline involves the translation of a closed AML model to a semantically equivalent Promela program. In this chapter we discuss what this translation looks like, what issues this translation may entail, and how we can solve these issues.

4.1 Translating AML to Promela

We start this chapter by defining a translation of closed AML models to Promela programs. As we shall see, the translation we define is intuitive, but incomplete. Specifically, the mapping we define here does not allow us to detect deadlocks. In the next section we rectify this by extending the mapping we define here.

Before proceeding, we must clear up some potentially confusing nomenclature. In AML the `state` keyword is used to define an explicit *state label*. However, a process' *state* consists of its current state label *and* the values assigned to its process variables. To avoid confusion we borrow terminology from STSs: a *location* is synonymous with a state label. Thus, a process' state consists of a location and a valuation for its process variables. The model's state is the state of all processes.

Promela programs and AML models are in some ways quite similar. Both involve processes with local variables communicating over channels using parameterized labels. But, there are also differences. A large part of the translation procedure is concerned with bridging these differences.

A relatively minor difference involves the scope of symbols. In AML, processes, locations, channels, and labels may share the same name. Processes, locations, channels, and labels may also share names with process variables and label parameter, but process variables and label parameters may not share names. Due to the syntax of AML, it is unambiguous at all times what type of construct a particular symbol refers to. In contrast, Promela requires symbols to be unique. For example, a channel name may not share a name with a process, nor may it share a name with a variable in the same scope. For this reason, we rename all elements of the AML model such that each element has a unique name, and use these names in the resulting Promela program.

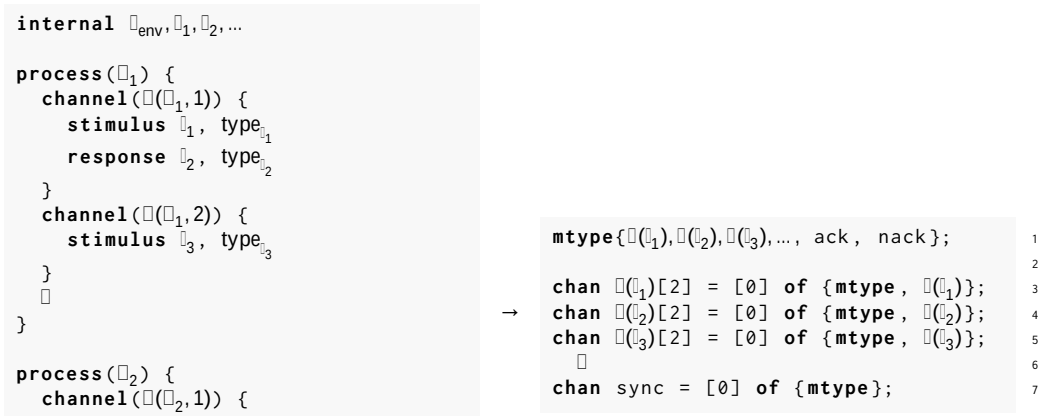

Figure 4.1 – Translating AML channels and labels of a closed model to Promela.

Channel semantics also differ between the two languages. In AML, messages of different types may be put on the same channel, but this is not allowed in Promela. In Promela, all messages put on a channel must be of the same type. For this reason we cannot simply copy or transform AML channel definitions into Promela channel definitions.

In the sequel we will present snippets of AML models where certain constructs have been replaced with variables or functions. We will then map these snippets to Promela, where we again make use of variables and functions. For clarity, we use Greek symbols for functions in the AML snippets. On the Promela side we use regular names for functions. We use Latin symbols for variables in both languages.

Let us start with the translation of the channels and the labels defined over them. We informally define a function operating on the AML code, □. This function maps a process and an index to a specific channel. On the Promela side we define three functions. A renaming function □, a channel function □, and a type function □. The □ function does what it says on the tin; it maps each name occurring in an AML model to a unique name. The □ function returns a unique channel name for each label, and the □ function prints a label's type.[1]

Figure 4.1 shows a skeleton of a closed AML model on the left side, and its translation to Promela on the right side. Promela uses constants of type `mtype` to define message constants. For each label in the AML model we define such a constant. When a process in the AML model sends or receives a message with a certain label, we use its corresponding message constant in Promela. For each label we create two channels of length zero. We explain why

[1] The □ and □ functions operate on a label *name* in Figure 4.1. In AML, it is possible to define multiple labels with the exact same name and parameter list, as long as they are assigned to different channels. These labels must be mapped to unique names in the Promela code. We assume that each label name is unique. However, in the real world this assumption can be, and often is, violated. Later in this section, when we apply the translation to the example model, we apply these functions to a label *object*, representing the entire label, not just the name.

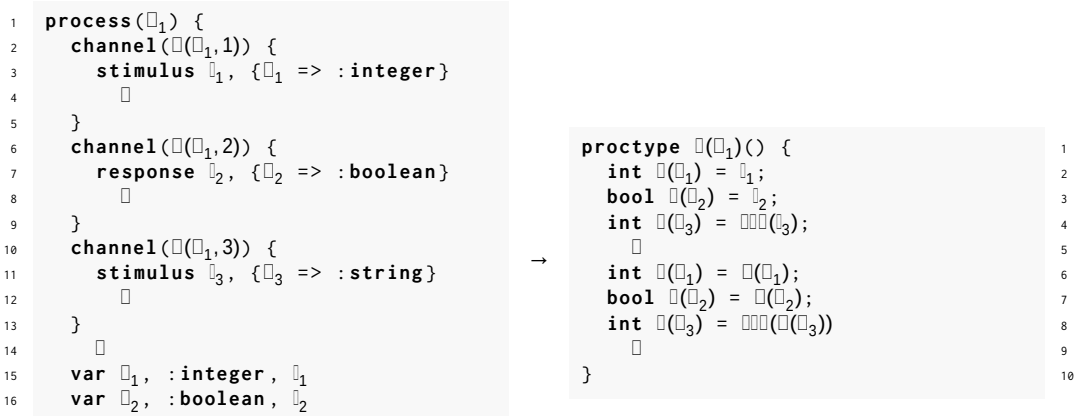

Figure 4.2 – Translating the initialization of AML processes to Promela.

we need two channels per label at a later point in this section. The type of each channel is an `mtype` followed by the types of the label parameters.[2] Note that the result of the function is empty for labels without parameters. We add a special `sync` channel. The `ack` and `nack` labels are the only labels that are communicated over this channel. We explain the need for this channel and its labels later in this section.

The `mtype` and channel declarations constitute the entire preamble. We can now translate the AML processes. Let us start with the initialization of each process. Figure 4.2 shows the translation. Each process variable and label parameter is initialized. The types we support are booleans, integers, and strings. Strings are not supported by SPIN. We encode each AML string as an integer in Promela. We provide more detail on this encoding procedure in Section 4.5. For now, we assume we have a string translation function , in addition to a default value function . The latter assigns a default value to each label parameter. Note that both the process variables and label parameters are initialized.

After the initialization follow the locations and their outgoing transitions. For the segments we will see later in this chapter, it is important to solidify our understanding of what an AML transition is. A transition is a tuple $(\square, \square, \square_{\square}, \square_{\square}, \square)$, where $\square$ is the source location, $\square$ is the target location, and $\square$ is the label. The symbols $\square_{\square}$ and $\square_{\square}$ stand for the constraint and update expressions respectively. A transition describes how the target location is reachable from the source location upon exchanging label $\square$ if $\square_{\square}$ holds, and how the process variables are updated by $\square_{\square}$. A $\square$-step is a special kind of transition: it is a transition with a $\square$-label, the unobservable label. A $\square$-step allows a process to advance to a new state without communicating with another process. Figure 4.3 shows common ways to define transitions in AML models. The top two fragments show how to define a single transition. The fragments on the lower row show how two transitions are defined. The difference lies in how the target location is defined. In the lower left fragment we define a target location for each option, while in the

[2] Technically, it is possible for labels with the exact same type to share a channel. That is, if label $\square_1$ and $\square_2$ have the same type, they may share a channel in the Promela code. However, we have decided to map each label to a unique channel for simplicity reasons.

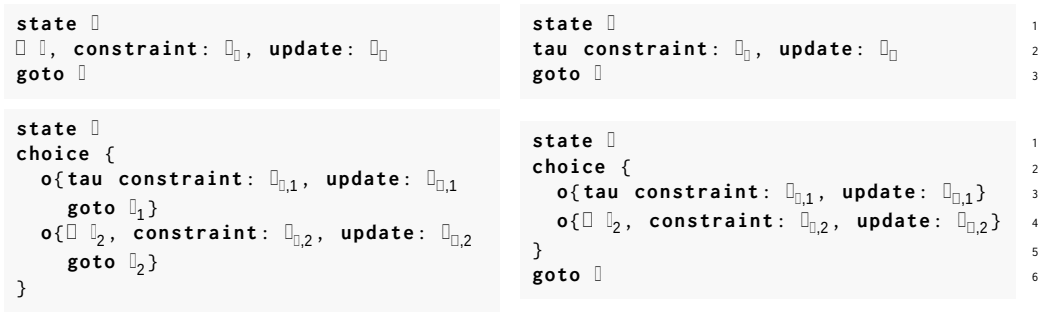

Figure 4.3 – Common ways to define transitions in AML. Note: □ □ { send, receive }.

right fragment we define a single target location for all options in the choice block.

The overall structure of the translation of an AML location is as follows. The location itself is renamed and used as a label we can jump to. If the location has no outgoing transitions we print `false` below the state label. In such a location the process is deadlocked; there is no way to advance to a new location.[3] If the location has one or more outgoing transitions, then we use an if-statement to non-deterministically select one of the enabled transitions. Figure 4.4 shows the overall structure. We use three functions, □□, □□, and □□, which expand the stimulus, response, and □-transitions respectively. As we shall see shortly, we need to group stimulus transitions together by their source location and label. The □□ function does exactly this.

Before we can define the translation of transitions, we emphasize two more important differences between AML and Promela. First, while both languages support pairwise synchronization, the semantics differ. In AML the evaluation of the constraints, the execution of the update functions, and the advancements to the next locations of both the sender and the receiver occur in a single non-divisible step. However, a synchronization in Promela just means the exchange of a message. This exchange may be subjected to constraints on both the sender side and the receiver side. Thus, the evaluation of these constraints and the exchange itself may occur in a single step. However, subsequent state changes are *not* part of the exchange. Second, label parameter values are implicit in AML, but explicit in Promela. The constraints of AML transitions implicitly define a set of label parameter mappings for which the constraint is valid given the current process variable valuation. We require a way to extract these implicit mappings, so that we can transcribe them to Promela, making them explicit. For now, we shall assume we have a function, □, which, given an AML response transition, returns a set of label parameter mappings for which the transition may be enabled.[4] In Section 4.4 we show how we implement such functionality.

While it is possible for a Promela synchronization to be constrained on both the sender side and the receiver side, AML constraints may refer to label parameters. Thus, the receiver can only determine if its constraint holds *after* it has received a label with its parameter values. For this reason, to simulate AML synchronizations in Promela, we have decided to make each synchronization a two-step process. First the sender picks a label and parameter

[3] This does not mean that the *program* is stuck, as other processes may still be able to perform actions.

[4] Whether the transition is actually enabled depends on the values of the process variables, and whether there is another process "ready" to exchange the message associated with this transition.

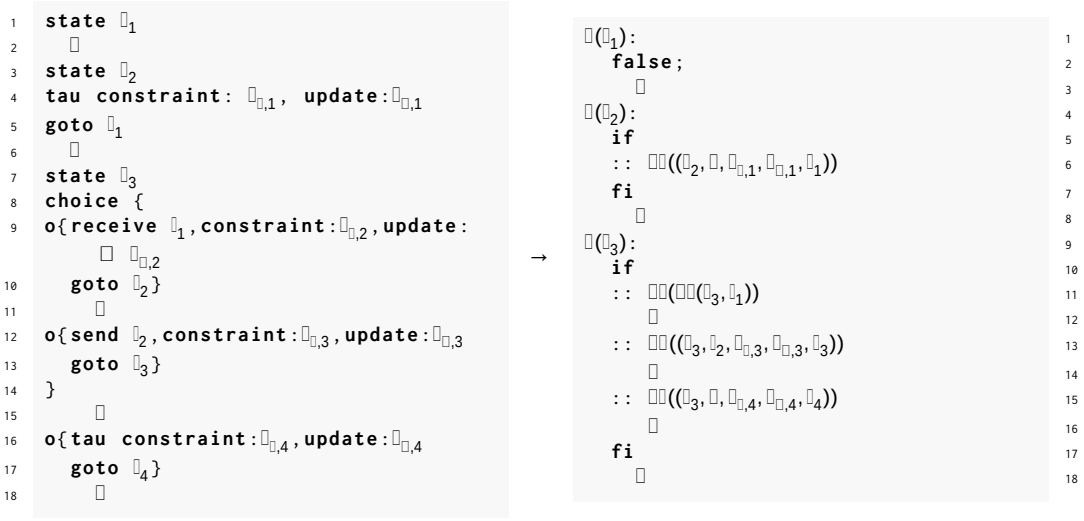

Figure 4.4 – Translation of AML states to Promela. The variables $\square_{\square}$, $\square_{\square}$ and $\square$ stand for arbitrary constraint expressions, update expressions, and transition targets respectively.

values to send, and determines whether its own constraint is valid. If so, it sends the label and parameters to the receiver. The receiver then determines whether one of the outgoing transitions from its current location is enabled for this message. SPIN will explore each enabled transition. At the end of each of these transitions, the receiver will send an acknowledgment to the sender, signaling that it has accepted the message and updated its local state. The sender then updates its own local state, completing the synchronization. If none of the receiver's transitions are enabled for the provided label and parameter values, then it returns a non-acknowledgment. The sender then knows it must abort this synchronization attempt. These acknowledgment (ack) and non-acknowledgments (nack) messages are sent over the sync channel, as seen in Figure 4.1.

We show the translations of transitions in Figure 4.5. Again, the AML skeleton is on the left, and the Promela translation is on the right. These translations essentially detail the $\square\square$, $\square\square$, and $\square\square$ functions. We see that the constraint and update expressions in the AML code make their way to the Promela side. However, since we previously we renamed the process variables and label parameters, we must substitute the old variable names for the new names in these expressions. This is what the substitution function, $\square$, does. Furthermore, the constraints of senders have to be transformed: when a label parameter name is equated to a process variable name, the label parameter name must be replaced by the process variable name, otherwise the constraint could never be true. This is what the normalization function, $\square$ does. Another function we require is the $\square$ function. This function returns a list of variables for a label.[5] When a process receives a label, the parameter values are assigned to these variables. The last additional function we require is the parameter reset function, $\square\square$. This function assigns the default values to a label's parameters, using the $\square$ function.

The first snippet shows the translation of a $\square$-step. This translation is relatively straightforward. We have a `d_step` that starts with a constraint. If the constraint is true, then the option

[5] The same assumption we made for the $\square$ and $\square$ functions applies here.

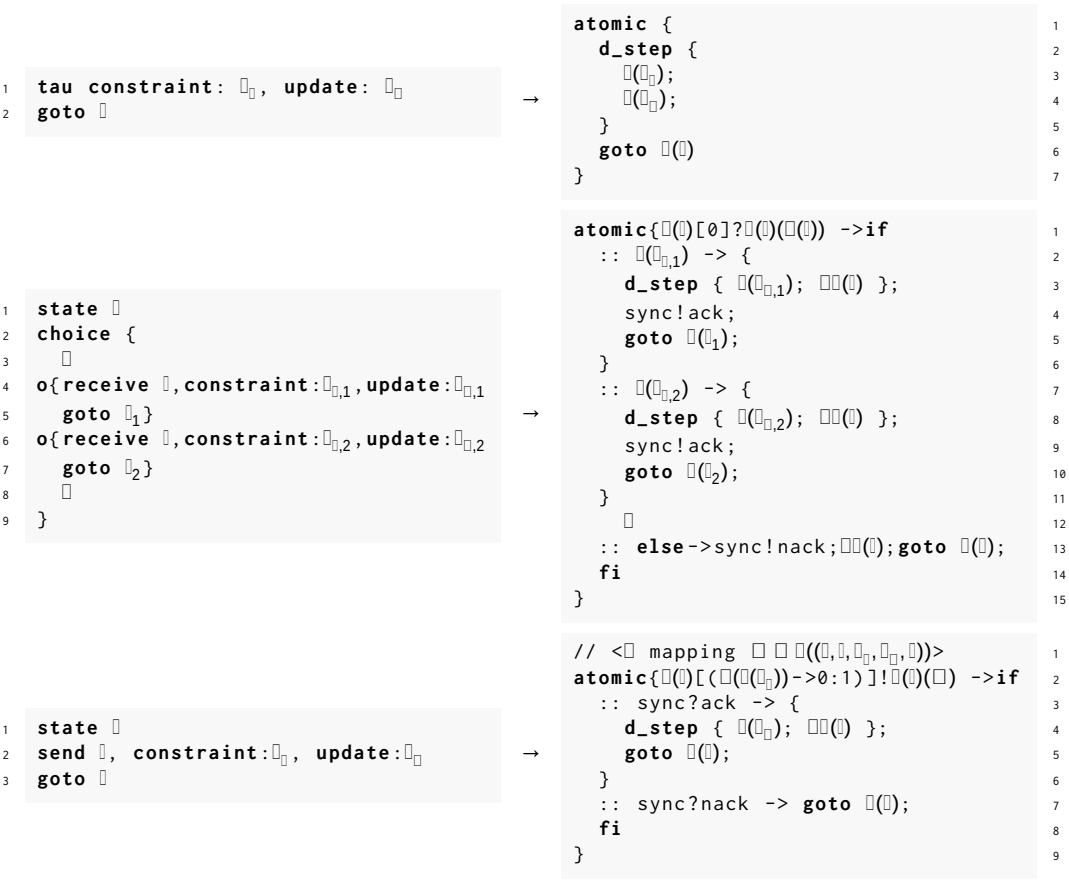

Figure 4.5 – Translation of AML transitions to Promela.

as a whole is enabled. When the option is executed, the constraint and update are executed as a single step. Without the `d_step` the execution of each statement in the atomic would have yielded a separate state, but with the `d_step` these steps are folded into one, reducing the overall state space. Note that `goto`-statements are not allowed in `d_steps`. However, as we are still in an atomic sequence, the advancement to the next state cannot be interrupted.

The translations of stimuli and responses are more interesting. We see that, for any label □, the receiver listens for messages with this label on channel □(□)`[0]`, but senders are capable of sending messages on both □(□)`[0]` and □(□)`[1]`. If its constraint is true, the sender picks the channel on index 0, otherwise it picks the channel on index 1. Thus, when the sender's constraint is false, the processes cannot synchronize. Note that, for every response transition in the AML model, there may be multiple options in the Promela program. A single AML transition may produce multiple Promela options as the transition may implicitly define multiple parameter mappings. Further, note that the mapping, □, the type function □, and the parameter expansion function, □, all respect the same ordering of the parameters. We also see that AML stimuli are grouped together by their label. When a message arrives, each of these stimuli are explored if they are enabled. If at least one stimulus is enabled, then the receiver sends an acknowledgment to the sender, upon which the sender may continue with its state update. Regardless of whether the synchronization attempt succeeded or not, the label parameter values are reset. This reduces the state space, as we do not arrive at a new

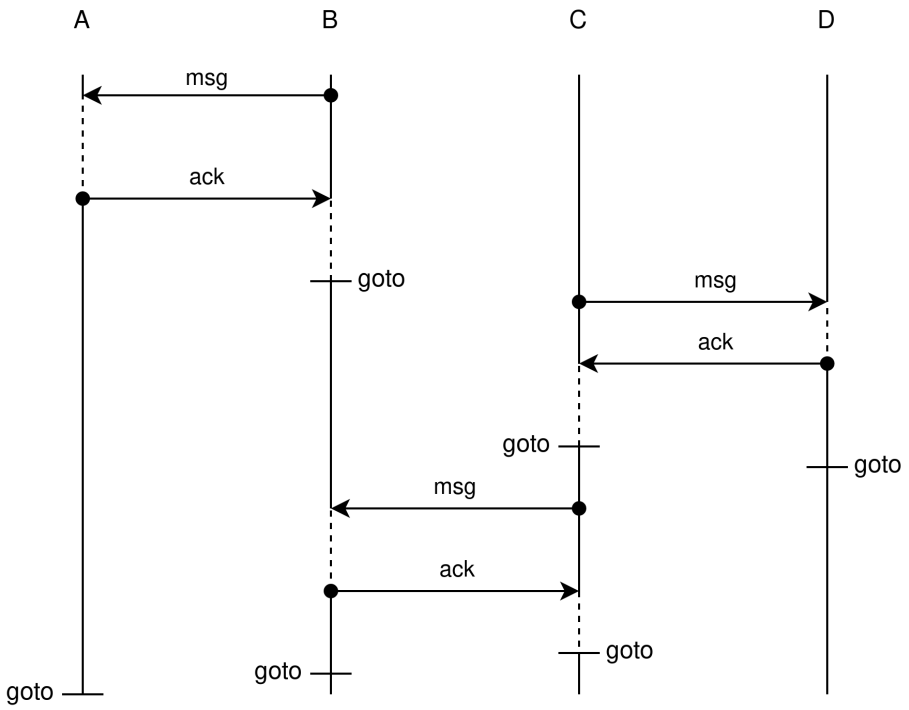

Figure 4.6 – Atomic synchronization between multiple processes.

state after a failed synchronization attempt due to the change in label parameter values. Thus, certain superfluous states are avoided. Grouping the stimuli together by label does mean that we must define the □□ function to work with sets of stimulus transitions. That is, the □□ function does not expand a single stimulus-transition, but rather a set of stimulus-transitions all with the same label, and leaving the same state.

While the exchange of acknowledgment messages solves the issue of receivers needing to receive label parameters before they can decide whether their constraint is valid or not, it does not grant us atomicity. We still need to manage access to the critical section. To achieve this we could make use of a semaphore or an arbiter. Alternatively, Promela statements sequences can be marked as atomic. A particularly useful feature is that a sender may transfer atomicity to a receiver by exchanging a message on a rendezvous channel.

In Figure 4.5 we mark each option as atomic. By doing so, we achieve atomicity. Though, with a caveat. Consider Figure 4.6. Here we find four processes on the horizontal axis, □, □, □, and □. The vertical axis represents time in discrete steps of the Promela program. Dashed lines indicate atomicity; the process is then in the critical section and no other process can do anything. Process □ enters its atomic block and sends a message to process □, immediately transferring atomicity to □. Process □ accepts the message. But, before it sends an acknowledgment, it first updates its local variables. These are simply assignments, and assignments in Promela never block. Once completed, □ sends the acknowledgment, returning atomicity to □. Process □ then updates its local variables. These are also all assignments, and therefore non-blocking. The last statement of each option is a goto-statement to the next location. Since this is also the last statement of the atomic sequence, process □ will

jump to its next location in one go. After that, atomicity is lost. We then see that processes □ and □ synchronize, even though process □ has not yet jumped to its next location. In addition, processes □ and □ synchronize before □ executes its goto-statement. Clearly, we are not achieving the goal of making sure *both* synchronizing processes advance to their next locations before the critical section is freed; there are states in the Promela programs that exist due to interleavings that do not occur in the AML model. However, this is not a problem. We can always return to a state that does exist in the AML model by executing all goto-statements of receivers that have not yet been executed. These are, in essence, just added □-steps. In Figure 4.6 we see that process □ loses atomicity, after which □ and □ synchronize, and then □ and □ synchronize. Process □ has only partially completed its synchronization with □. Thus we are in a state that does not exist in the AML model. By performing □'s goto-statement, however, we arrive in the state where we wanted to be in the first place; the state we arrive at after the synchronization of □ and □, followed by the synchronization of □ and □, and □ and □.

We can now apply the translation to our example model. Remember that the translation step is performed on the *closed* model. The closed model can be found on Page 19 (Chapter 3). First we create the message constants.

```
mtype {a, a_0, k, set_A, set_B, success_A, ACK, NACK};
```

We have two a labels, but they live on different channels. We must be able to tell the labels apart. For this reason one label may keep its name, while the other is renamed to a_0. In the original model, both process A and B defined an external set label. However, all labels defined on external channels have been renamed during the closure procedure. Since we can already tell label set_A apart from set_B, we do not need to rename these labels. The ACK and NACK labels are used for the synchronization mechanism. Next, we create the channels.

```
chan chan_a[2] = [0] of {mtype};
chan chan_a_0[2] = [0] of {mtype};
chan chan_k[2] = [0] of {mtype, int};
chan chan_set_A[2] = [0] of {mtype, int};
chan chan_set_B[2] = [0] of {mtype, int};
chan chan_success_A[2] = [0] of {mtype, int};
chan sync = [0] of {mtype};
```

After the channel translations, the processes follow. We only show the translation of process A here. We start with the declaration of the process itself, and the initialization of the local variables.

```
proctype A() {
  int i = 0;
  int reached_home_by = 4;
  int set_A_x = 0;
```

Remember that both the process variables and the parameters of all labels A can receive are initialized. In addition, note that the reached_home_by process variable is of type string in the AML model, but it is an integer here. After the process initialization come the locations and their outgoing transitions.

```
label_A_home:
  if
  :: atomic { chan_a[0] ? a -> if
    :: true -> {
```

```
5         sync ! ACK;
6         goto label_A_waiting;
7       }
8       :: else -> sync ! NACK; skip; goto label_A_home;
9       fi
10    }
11    fi
```

In the home location, process A can only accept a single message; an a label on channel int1. This is an unconstrained transition. Process A will always accept this label when process B sends it. This is reflected on line 4. Here we find an option that is always enabled. Note that, since the constraint is always true, the else branch is never reached. In addition, since the a label does not come with parameters, no parameters are reset. The next location is more exciting.

```
1  label_A_waiting:
2    if
3    :: atomic { chan_set_A[0] ? set_A(set_A_x) -> if
4      :: (set_A_x == 1 || set_A_x == 2) && i == 1 -> {
5        d_step {
6          i = set_A_x + 1;
7          set_A_x = 0;
8        }
9        sync ! ACK;
10       goto label_A__2;
11     }
12     :: else -> sync ! NACK; set_A_x = 0; goto label_A_waiting;
13     fi
14   }
15   :: atomic { chan_k[((i == i) -> 0 : 1)] ! k(i) -> if
16     :: sync ? ACK -> {
17       d_step {
18         i = i + 1;
19         reached_home_by = 1;
20       }
21       goto label_A_home;
22     }
23     :: sync ? NACK -> goto label_A_waiting;
24     fi
25   }
26   fi
```

Here we find two transitions. The first transition is a stimulus from the environment. It starts on line 3. Process A will only accept set_A messages when the x parameter equals either 1 or 2, and its process variable i is equal to 1. The second transition is a response. It starts on line 15. This transition is constrained. Based on the constraint, one of the two channels for label k is picked. The constraint here is always true. Note how the constraint was transformed from the AML model. In the AML model the constraint was x == i. This constraint tells us that the value of the x parameter must be equal to the value of process variable i. Thus, we can simply send i. The Promela constraint was produced by replacing all occurrences of x with i.

After receiving a set_A message in the waiting location, process A sends a success_A message to the environment. This implies that process A enters an implicit state after receiving the set_A message. This implicit state has been made explicit by the parsing process. We saw this implicit state in the closed model, where it was called gen_2. The Promela code for this location and its outgoing transition is constructed as before.

```
1  label_A__2:
```

```
  if
  :: atomic { chan_success_A[((i == i) -> 0 : 1)] ! success_A(i) -> if
    :: sync ? ACK -> goto label_A_set;
    :: sync ? NACK -> flags_success_A[0] = 1; goto label_A__2;
    fi
  }
  fi
```

At the last location we see the translation of a τ-step, concluding the translation of process A.

```
label_A_set:
  if
  :: atomic { chan_a_0[0] ? a_0 -> if
    :: true -> {
      d_step {
        i = 1;
        reached_home_by = 2;
      }
      sync ! ACK;
      goto label_A_home;
    }
    :: else -> sync ! NACK; goto label_A_set;
    fi
  }
  :: atomic {
    d_step {
      i == 0;
      reached_home_by = 3;
    }
    goto label_A_home;
  }
  fi
```

By applying the same transformation procedure to process B and the environment process we complete our translation. The complete Promela program can be found in Appendix B.

4.2 Detecting Deadlocks

The AML example model contains deadlocks. One such deadlock can be reached as follows: processes A and B exchange an a label on channel `int1`. Then process A sends a k message. This message comes with a parameter, x, which is equal to the value of the `i` variable of A. This variable is equal to 0. Both processes increment their respective `i` variables upon exchanging a k message, and return to their home locations. This cycle repeats twice more, and the processes exchange yet another a label. At this point, process A is at the waiting location, and B is at the comm location. The `i` variable of both processes is equal to 3. Process A cannot receive a set message, as the constraint for this transition cannot be satisfied. It is also incapable of sending a `k(3)` message, because process B will not accept it. Similarly, process B cannot send or receive any message in its current state, nor can it progress via a τ-step. Both processes cannot advance, and thus we find ourselves in a deadlock. However, if we provide SPIN the Promela program we constructed in the previous section it will not find any. The produced program will never be in a state where no action is enabled, and thus it is never in a deadlock. Instead, this deadlock will show up as several livelocks in the Promela program.

These livelocks are illustrated in Figure 4.7. The states of the model are illustrated as pairs of pairs. The first pair details the state of process A and the second pair shows the state of

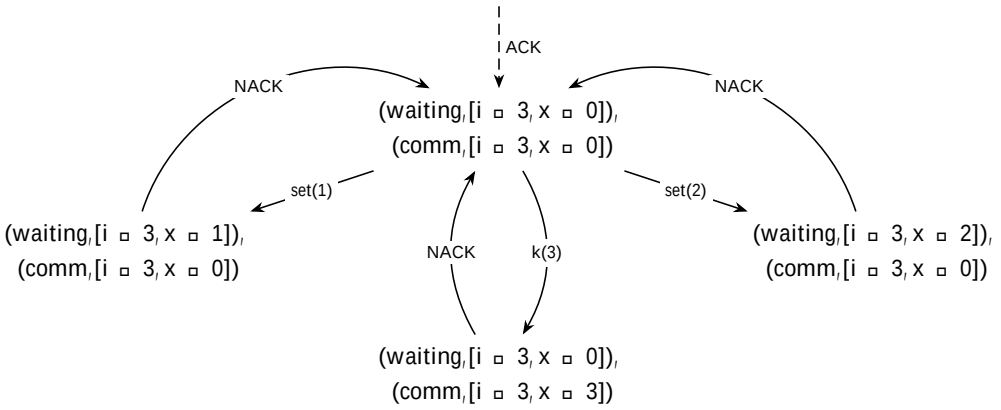

Figure 4.7 – Illustrating the livelock present in the Promela program produced by naively transforming the example AML model. Note that the `reached_home_by` variable has been left out.

process `B`. The first element of each pair shows the current location, and the second element shows the values for the local variables. We find three loops surrounding the start state. Two loops are started by a `set` message from the environment with two different parameter mappings. The other loop is started when process `A` sends a `k(3)` message to process `B`. What all three messages have in common is that the constraint on the sender side is true, but the constraint on the receiver side is not. The receiver returns a `NACK` message to the sender, upon which we arrive at a previously encountered state; the state before sending this particular message. In this state, the transitions for sending these three messages are enabled again. As a result, the system, globally, never halts and is therefore never in a deadlock.

One solution to this problem then, is to maintain a record of previously sent messages. If a message has already been sent before, and it was rejected, then it should not be sent again. Let us extend our mapping. For every label we create an array of bits. Each bit represents a flag. The function returns the name of this array for a given label. The length of each array belonging to some label is determined by the maximum number of messages with this label that can be sent from any location of the sender[6]. In our mapping, the flag count function, , returns this length. In Algorithm 1 we show how to calculate these lengths. We assume that the lengths are calculated and stored beforehand. The only retrieves the stored value.

We declare the flags globally, as part of our preamble. We also declare a variable used for iterating over the arrays. Figure 4.8 shows the new translation of the channel and label declarations. Let us now apply this new translation to the example model.

```
1 int flag = 0;
2 bit flags_a[1] = 0;
3 bit flags_a_0[1] = 0;
4 bit flags_k[1] = 0;
5 bit flags_set_A[2] = 0;
6 bit flags_set_B[2] = 0;
7 bit flags_success_A[1] = 0;
```

Both a labels have an arity of zero; they do not come with parameters. Therefore, we only

[6] Remember that for every label in a closed model, there is only one process that can send it, and only one process that can receive it.

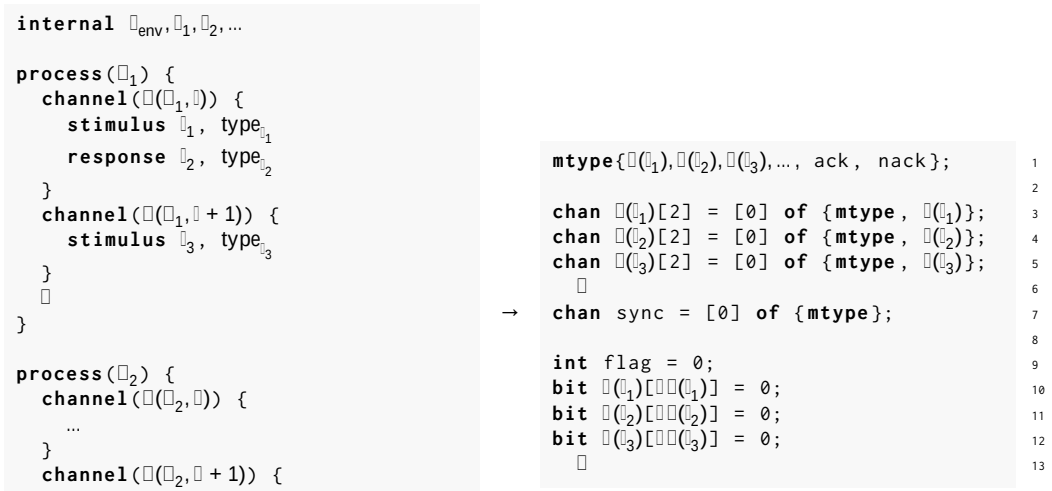

Figure 4.8 – Adding flags to the preamble.

require a single flag for each of these labels. The k label does have a parameter. There is, however, only one possible value for this parameter, and that is the value of the process variable i. The set_A label also comes with a single parameter. At the waiting location of process A we see that there are two value for this label, 1 and 2. At no other location can more than two set_A messages be sent. Thus, the length of the flag array for the set_A label is 2. Similarly, we find that we require two flags for the set_B label.

Next, we update the translation of the transitions. We must change the constraints on sending transitions such that the message can only be sent if the associated flag is not set. If it is set, the constraint must evaluate to false, disabling the transition. Furthermore, when a process changes state due to a successful communication or the execution of a □-step, the flags for certain messages must be reset. Since the process variables or location have changed, stimuli the process rejected previously may now succeed. The flags for these stimuli must therefore be reset. In addition, since the parameter values of response messages may depend on the

Algorithm 1 Counting the number of required flags for a particular label

```
1:  procedure COUNT FLAGS(process, label, mappings)
2:      count □= 0
3:      for state □ process do
4:          current_mappings □= □
5:          responses □= responses_with_label(state, label)
6:          for response □ responses do
7:              current_mappings □= current_mappings □ mappings(response)
8:          end for
9:          count □= max(count, |current_mappings|)
10:     end for
11: end procedure
```

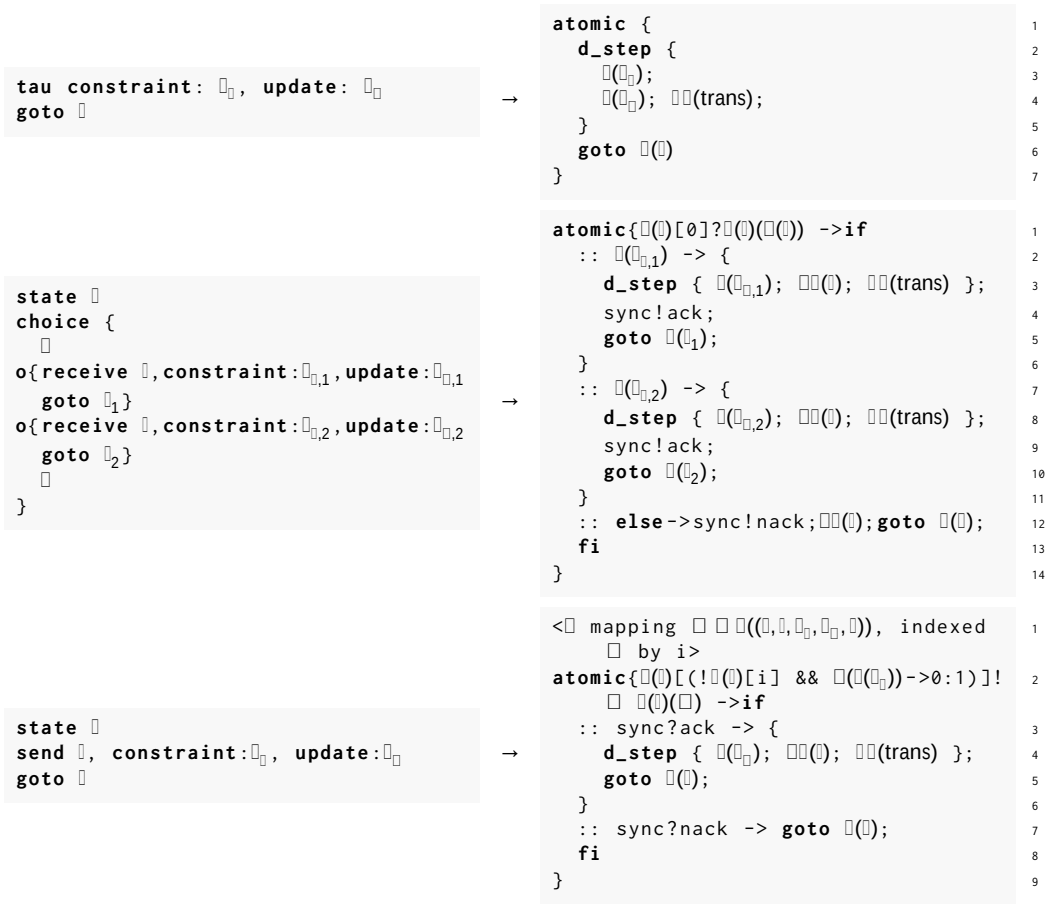

Figure 4.9 – Updated translation of AML transitions to Promela.

valuation of the process variables, flags for some of the responses must also be reset. We informally define a flag reset function, □□, that, given a transition, resets the appropriate flags. The general rule for determining which flags to reset is as follows: when a process changes state due to the execution of a □-step, then the flags for all labels belonging to responses and stimuli the process can respectively send and receive must be reset. When a process synchronizes with another process, it must reset the flags for all stimulus labels it can receive in all states, and it must reset the flags for all response labels it can send, except for the response labels addressed to the process it is synchronizing with. Figure 4.9 shows the updated mapping. We can now apply the new mapping on the example. Below we show the changes made to the `waiting` location.

```
label_A_waiting:
  if
  :: atomic { chan_set_A[0] ? set_A(set_A_x) -> if
    :: (set_A_x == 1 || set_A_x == 2) && i == 1 -> {
      d_step {
        for (flag in flags_a) { flags_a[flag] = 0; }
        for (flag in flags_a_0) { flags_a_0[flag] = 0; }
        for (flag in flags_k) { flags_k[flag] = 0; }
        for (flag in flags_set_A) { flags_set_A[flag] = 0; }
        flag = 0;
        printf("A[label_A_waiting]: chan_set_A ? set_A(%d) -> label_A__2\n",
            □ set_A_x);
        i = set_A_x + 1;
```

```
13          printf("A: [i = %d, reached_home_by = %d]\n", i, reached_home_by);
14          set_A_x = 0;
15        }
16        sync ! ACK;
17        goto label_A__2;
18      }
19      :: else -> sync ! NACK; set_A_x = 0; goto label_A_waiting;
20      fi
21    }
22    :: atomic { chan_k[(!flags_k[0] && (i == i) -> 0 : 1)] ! k(i) -> if
23      :: sync ? ACK -> {
24        d_step {
25          for (flag in flags_a) { flags_a[flag] = 0; }
26          for (flag in flags_a_0) { flags_a_0[flag] = 0; }
27          for (flag in flags_set_A) { flags_set_A[flag] = 0; }
28          for (flag in flags_success_A) { flags_success_A[flag] = 0; }
29          flag = 0;
30          printf("A[label_A_waiting]: chan_k ! k(%d) -> label_A_home\n", i);
31          i = i + 1; reached_home_by = 1;
32          printf("A: [i = %d, reached_home_by = %d]\n", i, reached_home_by);
33        }
34        goto label_A_home;
35      }
36      :: sync ? NACK -> flags_k[0] = 1; goto label_A_waiting;
37      fi
38    }
39    fi
```

Again, there are two transitions leaving the `waiting` location of process `A`. On line 22 we see a rendezvous expression where process `A` sends a `k` label to process `B`. The constraint has been changed to first check whether the flag for this message is set or not. If it is, then the constraint is false and the rendezvous is disabled. Both transitions reset certain flag arrays during a successful synchronization. When process `A` successfully receives a `set_A` message it must reset the flags for its stimuli: the flags for the `a` label on channel `int1` and `int2`, and the flags for the `set_A` label. The `set_A` label is sent by the environment process. Process `A` can send one label, `k`, to process `B`. Since process `B` plays no part in the synchronization between `A` and the environment process, the flags for the `k` label must also be reset. When process `A` successfully sends a `k` message it must still reset the flags for both `a` labels and the `set_A` label. However, since process `B` is now the synchronization partner, the flags for label `k` do not have to be reset by process `A`. Process `B` is responsible for resetting these flags.

We have added `printf` statements to both transitions. These statements are executed during a successful synchronization. We also use such statements in □-steps. When SPIN detects an error these `printf` statements are executed and put into the resulting trail file. There are two kinds of messages we include. On line 11 we see an example of the first kind. It reports a successful synchronization; process `A` is at location `waiting` and receives a `set` label with a particular value for the `x` parameter. On line 13 we find an example of the second kind. It simply reports the values of the process variables of process `A`. While this last kind is strictly not necessary to track how a deadlock state can be reached, it does make the process easier. In Section 2.3 we provide more detail on how the trails produced by SPIN with these `printf` statements explain how an erroneous state is reached by translating the trail steps back to AML-level steps.

An issue we have not tackled so far is starting the processes. A Promela process may be marked active, which means that it is running in the initial state. Alternatively, processes

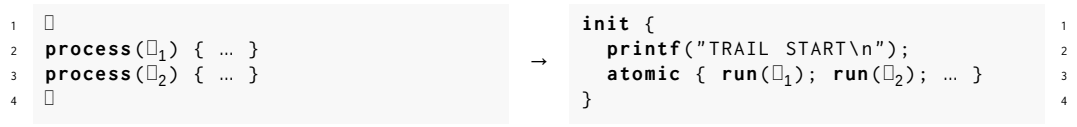

Figure 4.10 – Starting processes.

may be started using Promela's built-in `run` operator. To help us determine the start of the log, we print a specific message. As a consequence, we cannot mark processes as active. Instead, we use an `init` process that first prints this message, and then starts all the other processes. Figure 4.10 shows the translation. Applying this transformation to the example model produces the following:

```
init {
  printf("TRAIL START\n");
  atomic {
    run A();
    run B();
    run closure_env();
  }
}
```

With these additions to the transformation process described in the previous section we are fully capable of detecting deadlock states and reporting the steps required to get to them.

4.3 Flag Explosion

By maintaining a record of rejected messages we have solved the problem of certain AML deadlocks occurring as livelocks when SPIN performs its verification. The flag arrays we added enlarge the state vector of the overall program and may increase the size of the state space considerably. To illustrate this, we have created a second example AML model. This model is found in Listing 4.1. It is a rather small model, but also one of which we can easily increase the state space programmatically. The `counter` process can receive a `count` message with a parameter, `guess`, which should correctly "guess" the value of `counter`'s sole process variable, `i`. The environment sends a `count` label with one of three values for the `guess` parameter: 0, 1, or 2. Upon receiving a correct guess, the `i` variable is incremented. But, when `i` is equal to 2 and `counter` receives a correct guess, it will set `i` back to 0, and signal that `i` "maxed out". After that we have returned to the initial state of the model. Only one transition is enabled at any time, but the environment process in Promela must try to send all three of these guesses, since it does not know beforehand which guess is correct.

A bit string of length □ has $2^{□}$ different values. In this example we have three flags for the count label. Thus, we have $2^3 = 8$ possible values for the associated flag array. Of these possible configurations, 4 are reachable. Say `i = 0`. The only stimulus `counter` accepts is `count(0)`. However, instead of trying `count(0)` immediately, the environment process may try sending `count(1)` first. Or it might try `count(2)` first. It may also try both first. SPIN will explore all these orderings. Figure 4.11 illustrates this. In this figure we see three states SPIN will encounter as a result of unsuccessful synchronizations. Only `count(0)` results in a successful synchronization, after which we arrive in a new state. In this new state all possible orderings of sending `count(0)`, `count(1)`, and `count(2)` are explored again.

```
external 'default'

timeout 1

process('counter') {
  channel('default') {
    stimulus 'count', { 'guess' => :integer }
    response 'maxed_out'
  }

  var 'i', :integer, 0

  state 'home'
  choice {
    o {
      receive 'count', constraint: 'guess == 0 && guess == i', update: 'i = i +
          □  1'
      goto 'home'
    }
    o {
      receive 'count', constraint: 'guess == 1 && guess == i', update: 'i = i +
          □  1'
      goto 'home'
    }
    o {
      receive 'count', constraint: 'guess == 2 && guess == i', update: 'i = 0'
      send 'maxed_out'
      goto 'home'
    }
  }
}
```

Listing 4.1 – A worst case scenario when maintaining a record of rejected messages with flags.

The only configurations of the flag array that are not reachable are the ones where the flag for `count(0)` is set. If this process counted to 3 instead of 2, then we would have had 2^4 possible configurations of the flag array, of which 2^3 would have been reachable. In general, if a process is in a state where it can receive $□$ different stimuli, then there are $2^□$ possible configurations of the combined flags arrays[7]. If of these stimuli $0 \geq □ \leq □$ will result in an acknowledgment, then $2^{□-□}$ of these configurations are reachable. Therefore, if the number of reachable flag configurations is sufficiently high, then the number of states will explode. When we add a new possible guess to the `counter` process the size of the state space roughly doubles.

To verify this last point, we have created a template based on the model in Listing 4.1. The template allows us to verify the model for increasing values of $□$, where $□$ is the number of guesses `counter` may receive. Only one of these guesses is the correct one at any time. The results of the verification are listed in Table 4.1. The model checker was generated, compiled, and ran using the following settings:

```
spin -a model.pml
gcc -DMEMLIM=8192 -DNOBOUNDCHECK -DNOCLAIM -DSAFETY -DXUSAFE -O3 -o pan pan.c
./pan
```

We can see that the state space size increases exponentially. One would expect that the

[7] With "combined flag arrays" of a particular process we mean the concatenation of all flag arrays for labels that are a stimulus for this process. If flag array $□$ contains $□$ flags and flag array $□$ contains $□$ flags, then their concatenation contains $□ + □$ flags, and therefore has $2^{□+□}$ possible values.

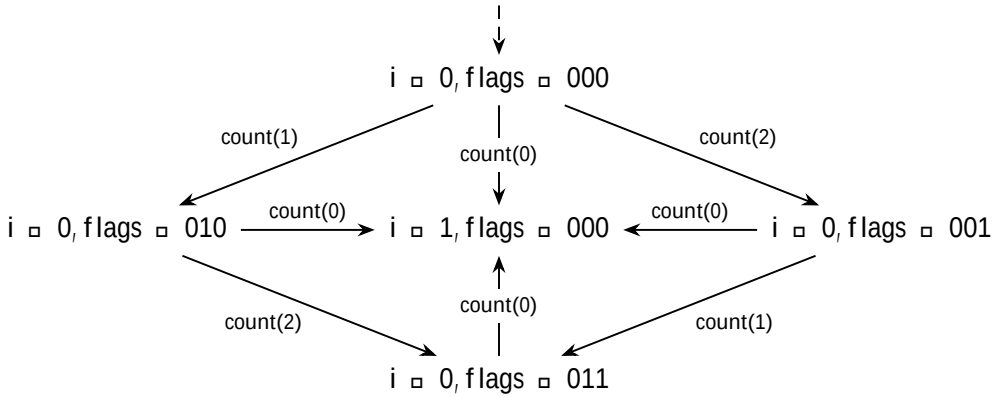

Figure 4.11 – State expansions starting at the initial state of the model found in Listing 4.1.

amount of states doubles for every guess. However, due to overhead introduced by the synchronization mechanism we find that the amount of states rises faster. As □ increases, the enlargement-factor seems to approach 2. Figure 4.12 confirms this. The state space sizes roughly lie on a straight line in logarithmic space, indicating exponential growth. Unfortunately, we did not have the computational capabilities to verify the model for □ > 20.

4.4 Parameter Extraction

At the start of this chapter we mentioned that the label parameter values are implicit in the AML model. When a sender and a receiver synchronize, the acceptable label parameters are determined by the constraints of both transitions. For both processes we can determine a set of parameter valuations that will satisfy the transition's constraint given the current process variable valuation. The intersection of this set contains exactly those valuations that satisfy both transitions. As we have seen, parameter values are explicitly sent in Promela programs. Thus, we must extract parameter values from the constraints of the input AML model. Determining these valuations is what we refer to as *parameter extraction*.

In the work of [41] a set of parameter valuations for a particular transition is determined by first finding all paths leading to the transition's source location. These paths are sequences of transitions. Then, for each path, the transition constraints are placed in a conjunction together with the constraint of the transition in question. Each conjunction is then fed into GNU Prolog's[8] finite domain solver. The result for one such conjunction is a set of valuations for the process variables and label parameters. The conjunction is satisfied for each of these valuations. Of course, we can then extract the valuations for the label parameters.

With the solution of [41] we consider the parameter extraction problem to be solved. For this reason, we have not attempted to come up with a better alternative. In Section 1.2 we discussed how we focus on a sub-language of AML by restricting the variable types, and update and constraint expressions. The allowed constraint expressions were deliberately chosen to ease the parameter extraction process. For stimulus transitions, each label param-

[8] `http://www.gprolog.org`

□	States	Depth
3	33	58
4	90	85
5	251	118
6	684	157
7	1805	202
8	4622	253
9	11535	310
10	28176	373
11	67601	442
12	159762	517
13	372755	598
14	860180	685
15	1966101	778
16	4456470	877
17	10027031	982
18	22413336	1093
19	49807385	1210
20	110100510	1333

Table 4.1 – The number of states of Listing 4.1 and the maximum search depth increase with □, where □ is the amount of possible guesses.

eter *must* be equated to a literal value. For response transitions, each label parameter *must* be equated to a literal value, *or* a process variable. Each constraint expression, regardless of whether it belongs to a stimulus or response transition, is a disjunction of conjunctions. Each conjunction defines a mapping for the parameters. Thus, for each constraint expression, we can determine a set of parameter mappings.

A transition's constraint may be empty. This implies that every possible label parameter value is allowed. For example, the constraints of all transitions of the environment process are empty; the environment is capable of sending and receiving all previously external messages. For every stimulus/response transition pair we require that at least one of them has a non-empty constraint if the label they exchange is parameterized. Since all transitions of the environment process have empty constraints, the constraints of external transitions with parameterized labels *must* be non-empty. Whether a transition of the environment process is enabled solely depends on the process the environment is trying to synchronize with.

Algorithm 2 shows the parameter extraction algorithm. It consists of two functions. The first function creates a function that, given a response transition, returns the set of all possible mappings. The second function determines all these mappings. A key understanding is that for any response transition, there may be multiple associated stimulus transitions in the other process. We extract a mapping from the response transition. Then, for each associated stimulus transition, we do the same. The stimulus mappings are combined into a single set. The intersection of the response mapping and the stimulus mappings contains exactly those mappings for which a response may succeed. Of course, none of these responses are

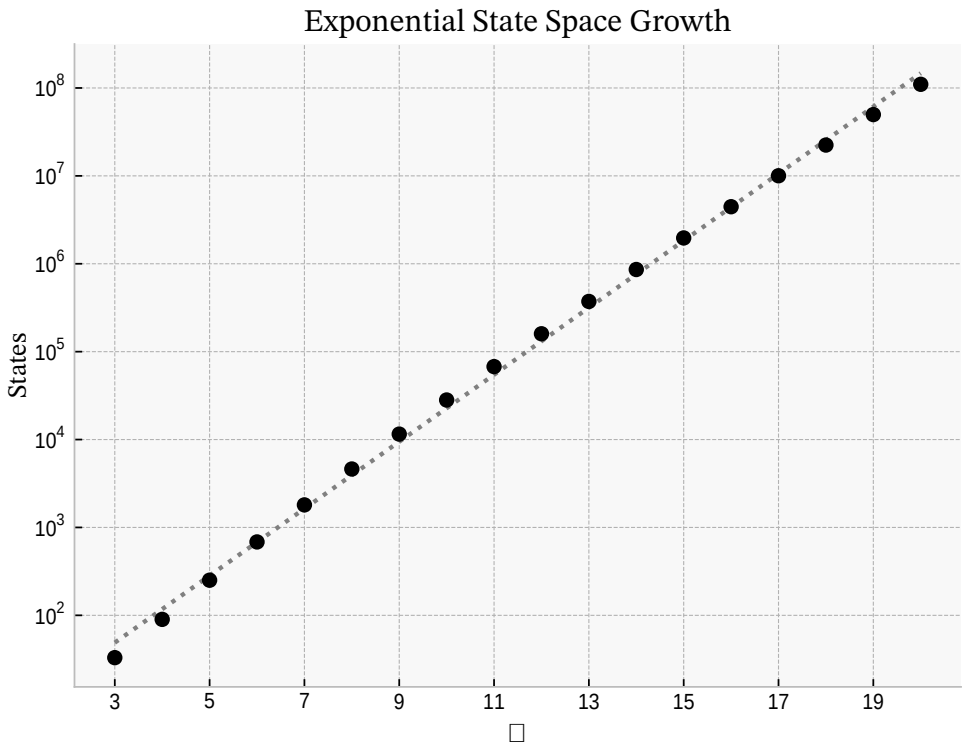

Figure 4.12 – The state space size of the model in Listing 4.1 increases exponentially for increasing numbers of □, where □ is the amount of possible guesses. Note that the □-axis is logarithmic and that the data points almost form a straight line.

guaranteed to succeed. Whether a synchronization will succeed still depends on the local state of both processes. Note that we use an intersect function, rather than the intersection operator (∩). This function is a bit special. Given a response mapping and a stimulus mapping, this functions considers the mappings equal if the stimulus mapping assigns the same literal values to a superset of the parameters to which the response mapping assigns literals to. When the mappings are considered equal, the response mapping is added to the result. For example, say we have response mapping {□ □ 3, □ □ □}, and stimulus mapping {□ □ 3, □ □ 7}. These two mappings are considered equal, and the former is added to the results.

4.5 String Extraction and Encoding

We have discussed adding flags to our Promela programs to make sure that certain kinds of AML deadlocks remain deadlocks in the translated Promela program. Furthermore, we have discussed the cost of this approach. There is still a gap in our translation description; we have not yet discussed how we are able to utilize strings in AML models, a type that SPIN does not support. This section fills that gap.

AML supports different data types than Promela. This raises a question: how can we still model check AML models that contain variables of a type not supported by SPIN? There are

Algorithm 2 Find all possible parameter mappings for each response transition in a model

```
1:  procedure CREATE PARAMETER MAPPINGS(model)
2:      mappings □= □
3:      for process □ model do
4:          for state □ process do
5:              responses □= state.responses
6:              for response □ responses do
7:                  response_mappings □= make_mappings(model, response)
8:                  mappings(response) □= response_mappings
9:              end for
10:         end for
11:     end for
12:     return mappings                          □ Function: response → mapping
13: end procedure
14:
15: procedure make_mappings(model, response)
16:     response_mappings □= extract_params(response.constraint)
17:     stimuli_mappings □= □
18:     stimuli □= stimuli_for(response, model)
19:     for stimulus □ stimuli do
20:         stimulus_mappings □= extract_params(stimulus.constraint)
21:         stimuli_mappings □= stimuli_mappings □ stimulus_mappings
22:     end for
23:     if response_mappings = □ □ stimuli_mappings = □ then
24:         return {□}                           □ Singleton with empty mapping
25:     else if stimuli_mappings = □ then
26:         return response_mappings
27:     else if response_mappings = □ then
28:         return stimuli_mappings
29:     else
30:         return intersect(response_mappings, stimuli_mappings)
31:     end if
32: end procedure
```

multiple possible solutions. We provide one together with an implementation: encoding.

Data types like bools and integers are finite: a data type that is □ bits wide consists of up to $2^{□}$ possible values. Some data types are, at least theoretically, of infinite size, like strings and arrays. If an AML data type is finite, then we can simply replace the values of this type with values of another type of equal or greater size. For example, say we have an AML data type consisting of $2^8 = 256$ possible values. We can replace these values with values from a Promela type containing at least 256 values. Integers form the widest available scalar data type in Promela, at 32 bits wide. If the AML data type consists of more than 2^{32} values, then we can simply use a combination of multiple variables[9] that together contain enough values.

It may not be necessary to use a combination of variables to represent an AML data type with more than 2^{32} values. Often only a subset of these values occur during the lifetime of the system. By determining the size of this subset statically, we may be able to represent this AML data type with fewer or smaller variables, thus keeping the memory footprint minimal. In

[9] Given a datatype □ and a datatype □, the combination of these two types is the Cartesian product between the two. The size of this type is then |□| □ |□|.

addition, it allows us to encode variables from a data type of infinite size, as long as the subset of occurring values is finite. Take strings for example. Theoretically, an infinite number of strings exist. However, if we can determine all strings occurring during the lifetime of the system, and this amount is finite, then we may still utilize the same encoding strategy.

As a proof of concept, we have added support for AML strings. We analyze the model and extract all string values. This is possible due to the restrictions we have placed on the constraint and update expressions. As stated in Section 1.2, strings only occur in update and constraint expressions as either string literals or references to other variables of type string. Thus, we can find all string values by investigating the initial values for the process variables, and extracting all string literals from the update and constraint expressions. Each string value is assigned to a unique integer. When we write the Promela code, each string value is replaced by its associated integer. Since the number of strings literals in an AML model is typically quite small, we have not implemented using multiple variables to encode more than 2^{32} strings.

4.6 Reduction Through Extension

In Section 4.2 we extended our mapping by adding flags. With these flags we prevent certain kinds of deadlocks from occurring as livelocks. By doing so, we have added extra information to the Promela programs, information that was not present in the AML model. As we saw in Section 4.3, this extra information may dramatically increase the state space size. Thus, we have an incentive to remove or at least reduce this effect.

SPIN supports the embedding of C code in Promela programs [44]. This feature allows users to change the behavior of SPIN without changing SPIN itself. We have attempted to use this feature to “hide” the flags. That is, using these C-extensions, we have instructed SPIN to not consider the flags part of the state descriptor, while still using them to maintain a record of failed synchronization attempts. While initial results were promising, we have not been able to make this addition work reliably. Still, we consider changing the behavior of SPIN to better suit our needs, either through these C-extensions or changing SPIN itself, a viable future research endeavor.

Chapter 5

Results

In this chapter we evaluate our prototype. We compare the Promela programs produced by our pipeline with reference solutions for well-known computer science and engineering problems. Furthermore, we have created AML models of which we can easily calculate the number of states and deadlocks their underlying IOSTS contains based on a few parameters. We show that the Promela programs produced by our pipeline contain the exact same number of deadlocks, providing confidence in our mapping.

5.1 Unit Tests

We have designed and built a pipeline that transforms open AML models to closed Promela programs. Before we proceed with other experiments, we want to provide confidence in our translation. In particular, we want to show that we are able to find all deadlocks present in an AML model. We have created three experiments that are akin to unit tests; they do not prove that our translation is correct, but they do serve as sanity checks.

We have created three templates that produce AML models. Each model follows the same overall structure. Figure 5.1 shows an example of one particular model. On the left we find the AML code, and we find the graphical representation of the IOSTS it models on the right. It is a model containing a single process with a single process variable, `i`, and four locations. In the initial location, one of three follow-up locations is selected non-deterministically. In this selected follow-up location the process may only receive `inc` messages from the environment. Variable `i` is incremented upon receiving such a message. These receive-transitions are constrained; the process and the environment can synchronize only when `i < 3`. These constraints thus impose an upper bound on `i`. Note that, once a follow-up location has been selected in a process' initial state, we cannot leave this location. Thus, as `i` reaches its upper bound, we arrive at a deadlock.

The three templates differ only with respect to the `inc` label. The first template produces models where this label is not parameterized. The model in Figure 5.1 is an example of such a model. We call these models "simple receivers". In models produced by the second template, the `inc` label is parameterized by a single integer, called `x`. A process and the environment

```
external 'ext'
process('proc1') {
  channel('ext') {
    stimulus 'inc'
  }
  var 'i', :integer, 1

  state 'start'
  choice {
    o { goto 'p1' }
    o { goto 'p2' }
    o { goto 'p3' }
  }
  state 'p1'
  receive 'inc', constraint: 'i <
      □ 3', update: 'i = i + 1'
  goto 'p1'

  state 'p2'
  receive 'inc', constraint: 'i <
      □ 3', update: 'i = i + 1'
  goto 'p2'

  state 'p3'
  receive 'inc', constraint: 'i <
      □ 3', update: 'i = i + 1'
  goto 'p3'
}
```

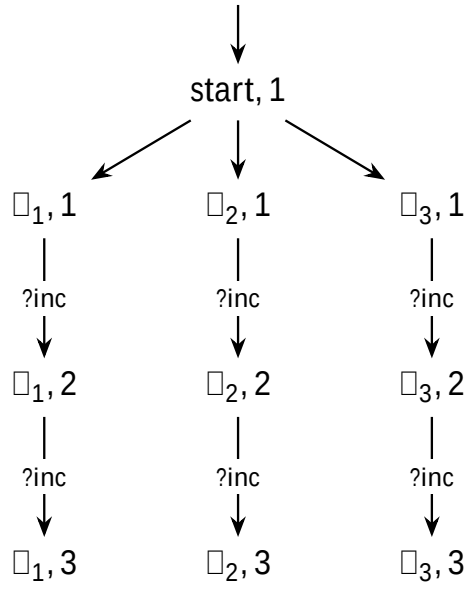

Figure 5.1 – Simple receiver for □ = 1, □ = 3, and □ = 3. The second element of each tuple denotes the value of `i`.

can only synchronize when `i == x`. We call these models "symbolic receivers". In models produced by the third template, the `inc` label is again parameterized by integer `x`. However, we reverse the direction of the label; these models *send* `inc` labels instead of receiving them. When sending an `inc` message, `x` takes on the value of `i`. We call these models "symbolic senders". In Appendix C.1 we provide examples of each kind of model.

Because each model follows the same overall structure, the IOSTSs they model also follow the same overall structure. We have specifically designed these models such that we can easily calculate the number of states and deadlocks for the underlying IOSTSs. Each template takes three parameters; □, □, and □, which stand for the number of processes, the upper bound for `i`, and the number of locations (not counting the initial location), respectively. All processes are identical: in the initial state one of the □ locations is picked. In this next location `i` is incremented until it hits its upper bound, □, as part of a synchronization with the environment. With these parameters, we can state the formulas to calculate the states and deadlocks. We assume □ > 0 for both formulas. For all □ processes, there are □ + 1 locations, and `i` may take on □ values. The value of `i` can only be changed in one of the □ locations. Thus the number of states of the underlying IOSTS for each model produced by any of the three templates is as follows:

$$\text{states} = (\square \square \square + 1)^{\square}$$

We arrive in a deadlock when `i` hits its upper bound in all processes. This can occur in any

Locations	Projected States	Simple States	Symbolic Receive States	Symbolic Send States
2	343	4821	65895	937
3	1000	15352	217339	2806
4	2197	35339	509255	6259
5	4096	67812	987747	11782
6	6859	115801	1698919	19861
7	10648	182336	2688875	30982
8	15625	270447	4003719	45631
9	21952	383164	5689555	64294
10	29791	523517	7792487	87457

Table 5.1 – State space sizes for the simple receivers, symbolic receivers, and symbolic senders with $\square = 3$, and $\square = 3$. The projected states column lists the number of states of the underlying IOSTS, calculated using the previously defined formula.

of the $\square$ locations of all $\square$ processes. Thus, the number of deadlocks is as follows:

$$\text{deadlocks} = \square^{\square}$$

For this experiment we have generated and verified AML models produced by these three templates for varying values for $\square$, $\square$, and $\square$. The simple receivers and symbolic senders were verified for $1 \leq \square \leq 4$, $1 \leq \square \leq 3$, and $1 \leq \square \leq 10$. The state spaces of the symbolic receivers grow much faster compared to the other models when changing any variable. We were unable to verify these models for $\square > 3$ due to starvation of computational resources. Each generated Promela program was verified using the following settings:

```
spin -a model.pml
gcc -DMEMLIM=7800 -DXUSAFE -DNOBOUNDSCHECK -DNOFAIR -DSAFETY -O3 -w -o pan pan.
    □ c
./pan -c0 -m50000000
```

The `-c0` option tells the verifier to not stop the verification when it finds an error. It must continue until it has explored all states.

For each model kind we were able to find the exact number of deadlocks for all parameter combinations. No deadlocks were added, and none were missed. Table 5.1 lists state space sizes for all program kinds. We fix $\square = 3$, and $\square = 3$. Only $\square$ varies in this table. We saw in Section 4.1 that our Promela synchronization mechanism allows for interleavings that do not exist in the AML models. In addition, in Section 4.2 we added flags to the generated Promela programs, further enlarging their state spaces. In line with our expectations then, the Promela programs produced by our pipeline for these AML models contain more states than the IOSTSs these AML models model. We also see large disparities between the different model kinds. The symbolic receivers in particular, grow much faster than the others. These differences stem from how each model interacts with the environment.

Let us first compare the symbolic receivers with the symbolic senders. The symbolic receivers have the largest state spaces, while the symbolic senders have the smallest state spaces. Let us assume $\square = 1$. After choosing one of the $\square$ follow-up states, a symbolic receiver can

only synchronize with the environment when the environment sends an `inc(x)` message. Parameter `x` must be equal to the current value of `i`, and `i` must be unequal to □. The environment process is unaware of the current value of `i`. As we saw in Section 4.3, this means that the environment process contains a self-loop for each possible guess for `i`. Thus, the environment process can send □ `inc(x)` messages. In a non-deadlocked state, only one of these messages will be accepted, while in a deadlocked state, none of these messages will be accepted. In contrast, symbolic senders send `inc(x)` messages to the environment. The value of `x` is equal to the current value of `i`. Since the environment accepts anything, this synchronization never fails. Consequently, symbolic receivers interact much more with the environment than the symbolic senders, and the flags have a stronger effect on the size of the state space.

The difference between the simple receivers and the symbolic senders stems from the direction of the interaction with the environment. Let us again assume □ = 1. Consider the situation where a simple receiver successfully synchronizes with the environment, and, as part of this synchronization, `i` hits its upper bound. In the underlying IOSTS this is a deadlocked state. However, in the Promela program the environment must attempt to send an `inc` message once more. This synchronization attempt will then fail, and the associated flag will be set. Only then can the verifier conclude that a deadlock has been reached. In contrast, when a symbolic sender reaches this state, it will simply stop interacting with the environment. Because `i` has reached its upper bound, the symbolic sender cannot send any message to the environment.

With these three experiment we have found evidence that Promela programs produced by our pipeline do indeed find all deadlocks. We also saw that our mapping does introduce superfluous states, as expected. How many superfluous states are added, however, does depend to some degree on particular modeling decisions.

5.2 The Bounded Retransmission Protocol

The Bounded Retransmission Protocol (BRP) is a communication protocol designed for exchanging messages over an unreliable medium [45]. Each message is split up into a list of □ chunks that are sent in order. When the sender sends a chunk, it waits for an acknowledgment from the receiver. If an acknowledgment does not come in time, then the sender sends the same chunk again. The amount of attempts the sender can make is bounded by □, hence the name. If the bound is reached, then the message transmission is aborted. Each message contains an alternating bit, providing the receiver with enough information to determine whether it received new or old data. Figure 5.2 provides an overview of the protocol. The sender process, □, and the receiver process, □, communicate via channels □ and □. Sender □ receives chunks from the environment on $□_{in}$, and signals whether the message transmission was successful or not on $□_{out}$. Receiver □ sends each chunk it receives along with a status message to the environment on $□_{out}$. For more information regarding the protocol, we refer the reader to [45].

In [45] several properties of the protocol are verified. This verification was performed in two steps. Properties regarding the timing aspects of the protocol were verified using UP-

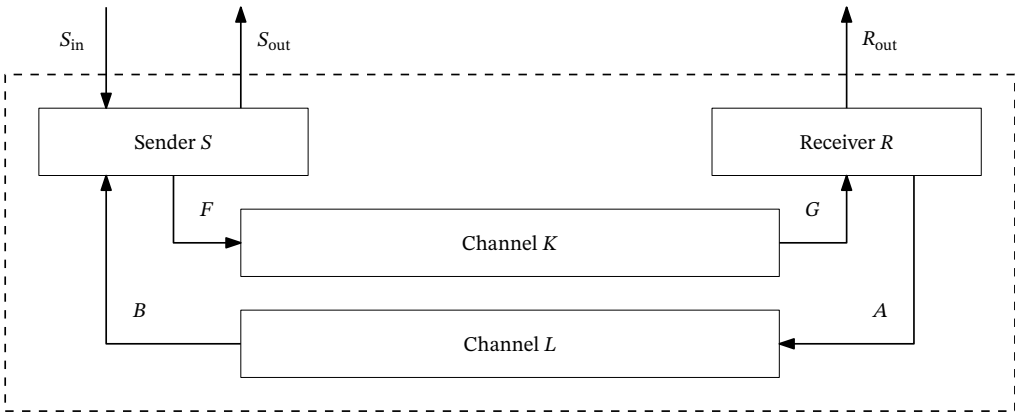

Figure 5.2 – An overview of the BRP. Reproduced from [45].

PAAL [46], and other properties were verified with SPIN. We base our AML implementation of the BRP on these Promela programs. We validate our prototype in two steps. First, we verify that the Promela program produced by the prototype satisfies the same properties as the Promela programs provided by [45]. Second, we compare the state space growth rate of the programs produced by the pipeline with the programs of [45].

The work of [45] provides two Promela programs that model the BRP; a standard one, and an optimized one. These programs differ in two ways: how byte sequences of random length are generated, and how the channels and are modeled. The authors of [45] note that the considerable state space of the standard program is in large part caused by the way the chunk sequences are generated. The optimized version contains a method that produces a far lower amount of random sequences, and therefore has a much weaker impact on the overall state space. This new method does not influence the correctness of the verification. The optimization of channels and is more subtle. In the standard program, and are modeled as processes. When sender sends a chunk, it puts the chunk, along with some meta-data, on channel . Process receives and stores this message. Process can then pass the message on to the receiver by putting the exact same chunk and meta-data on channel . However, it can also send a timeout message to the sender instead, modeling the timing aspects of the protocol. Similarly, receiver sends acknowledgments to the sender for each chunk it receives. It exchanges the acknowledgment with process over channel . Process can then either relay this acknowledgment over channel , or send a timeout. The optimized program replaces process and with buffered channels, each holding at most one message. A daemon process may “steal” messages from these channels, and send timeout messages instead.

The state space reduction offered by the optimized program is considerable. Our initial exploration showed that the standard program’s state space increases dramatically for each increase of , the number of chunks. This increase is so severe, that we were unable to perform the verification on the standard program with > 3, regardless of the value of the maximum number of retransmissions, . Consequently, basing our model on the standard program, and comparing our solution with it, would not yield much meaningful data. Instead, we would rather compare our model with the optimized program. However, while we can

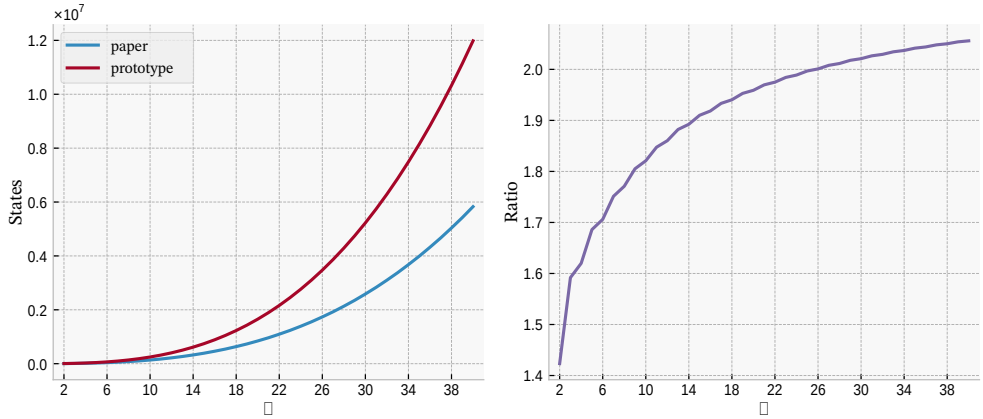

Figure 5.3 – Comparing the state space of the edited Promela program from [45] with the state space of the Promela program produced by the pipeline, for increasing values of (left). On the right, the ratio between these two sizes is plotted.

replicate the chunk generation method, the prototype only produces non-buffered channels. Thus, we cannot model channels and as actual channels. Instead, we must model them as processes, like in the standard Promela program.

Another modeling difficulty comes from the environment process. Both the standard and optimized program contain one. It is responsible for generating the chunks, communicating with the sender and receiver, and verifying a number of properties once the message transmission has finished, which occurs when all chunks are successfully delivered, or when the transfer is aborted. The environment process our prototype generates is only responsible for sending and receiving all relevant inputs and outputs to all other processes. It does not generate chunk sequences, nor does it verify any property.

For our first experiment we verify that the Promela program produced by our pipeline satisfies the same properties as the programs from [45]. To perform this experiment, we created an AML model that closely follows the standard program. This model can be found in Appendix C.2. Every interaction with the environment in the standard program is also an interaction with the environment in the AML model. We then produced the Promela program for our AML model, and replaced its environment process with the one from the optimized program. Some changes had to be manually made to glue it all together:

- Labels used for communicating with the environment had to be renamed,
- After each communication from the environment, specific flags had to be reset.
- The environment generates a random sequence of chunks (bytes), and writes these to a global array. The sender must read from this array when it sends data to the receiver,
- As part of our two-step synchronization process, the environment must send and receive acknowledgments for each message it receives and sends respectively,

With these changes in place we have a valid Promela program. This program does indeed satisfy the same properties as the standard and optimized programs from [45].

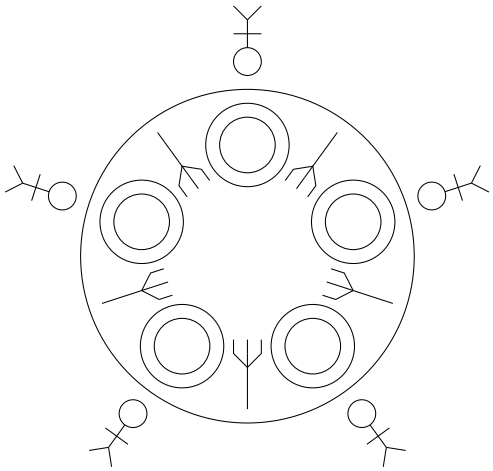

Figure 5.4 – Artist impression of the dining philosophers problem.

We can now proceed with our second experiment. We compare the state space size of the Promela program produced by our prototype, with the state space size of the optimized program, for differing values of □. We fix □ at □ = 3. To perform this experiment, we created templates for both our AML model, and the optimized program. We base our AML model template on the model from our previous experiment. This time, however, we did not replace the environment of the produced Promela program. Since the environment processes our prototype generates are incapable of generating random data, we instead make the sender responsible for this task. The generation method is based on the one found in the optimized program. Channels □ and □ are, again, modeled by processes. In Appendix C.2 we show an AML model produced using this template. Environments generated by our prototype also do not verify properties. In the interest of fair comparison, we remove the property verification from the optimized program template.

Figure 5.3 shows the results of this experiment. The left graph shows how the state space size increases as □ increases for both the edited program from [45] (paper), and the program produced by the pipeline (prototype). The graph on the right shows the ratio between these two sizes:

$$\text{Ratio} = \frac{\text{number of states in paper program}}{\text{number of states in prototype program}}$$

We find that the prototype program always contains more states than the paper program. However, as □ increases the ratio stabilizes. Unfortunately, we cannot verify both programs for □ > 40. Regardless, we can clearly see that, although the prototype does introduce superfluous states, the overhead is small enough that we are still able to adequately verify the BRP.

5.3 Dining Philosophers

Let us proceed with a venerable but still relevant problem: the dining philosophers problem. Originally called “the problem of the dining quintuple” by its inventor, Edsger Dijkstra, in 1965, and later renamed by Hoare [47], the dining philosophers problem is a non-trivial exercise in reasoning about deadlocks. It is a problem that scales; we can vary the number of

Reference		
□	States	Depth
3	28	32
4	82	113
5	244	336
6	730	997
7	2188	2932
8	6562	9937
9	19684	29608
10	59050	92229
11	177148	288962
12	531442	873007
13	1594324	2654568
14	4782970	8099483
15	14348908	24448304

Prototype		
□	States	Depth
3	163	191
4	680	646
5	2832	2085
6	11699	6152
7	48092	18718
8	197237	64782
9	808431	193401
10	3314484	616133
11	13596974	1952223

Table 5.2 – Tables comparing the state space sizes and maximum required search depth for the reference and prototype programs for the dining philosophers problem.

philosophers, □, which influences the state space size. In this experiment, we compare the state space sizes of Promela programs produced by our pipeline with the state space sizes of reference solutions, for varying values of □.

We start with a short description of the dining philosophers problem. Five silent philosophers sit around a round table. In front of each philosopher is a plate with an infinite amount of spaghetti. To the left and right of each philosopher lies a fork. Figure 5.4 shows this situation. A philosopher either eats or thinks. To eat spaghetti, a philosopher must use both forks adjacent to them. Consider what would happen if all philosophers were instructed to behave as follows:

1. Think until the left fork is free. When it is free, pick it up,
2. Think until the right fork is free. When it is free, pick it up,
3. Enjoy dinner,
4. When done eating, put down the left fork,
5. Put down the right fork,
6. Start from the beginning.

A deadlock is possible with these instructions. If all philosophers pick up the fork on their left, they will all be stuck thinking. No philosopher has access to the fork on their right, and therefore nobody can start eating and later put down their forks. One solution to this problem is to make one philosopher pick up their forks in the opposite order. This way, there is at least one philosopher capable of having dinner and finishing it at any time.

For this experiment we created two templates; one template produces AML models, and the other produces Promela programs. Both templates take □ as their sole parameter. We call the Promela programs produced by the pipeline for these AML models “prototype programs”,

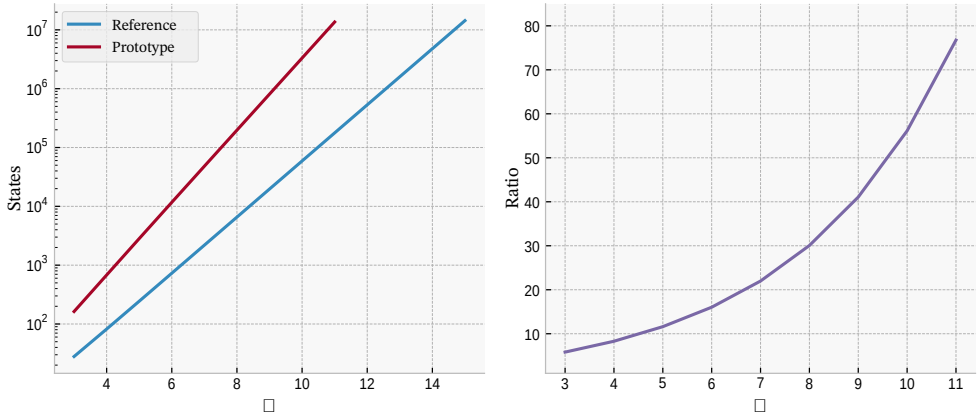

Figure 5.5 – Comparing the state space sizes of the two programs. The graph on the left shows the growth for both programs on a logarithmic scale. The graph on the right shows the ratio between these two sizes.

and we call the Promela programs produced by the other template "reference programs". In the interest of fair comparison, both kinds of programs are structurally similar. The reference programs represent programs one would make for the dining philosophers problem manually. In Appendix C.3 we have listed both programs for $\square = 3$.

Table 5.2 shows the results for the reference and prototype programs. We see that the prototype programs have much larger state spaces than their reference counterparts. In addition, they require a deeper search to validate all states. Since both the reference programs and the prototype programs follow the same structure, we see evidence that our mapping does indeed introduce superfluous states. In addition, we see that the state spaces of the prototype programs grow faster: for every increase of $\square$, the state space roughly quadruples. In contrast, the reference program state space sizes roughly triple for each increase of $\square$.

This difference in growth rate is further illustrated in Figure 5.5. In the left graph, we see the state space sizes plotted on a logarithmic scale. The line representing the state space sizes of the prototype programs has a higher slope than the other line, thus showing that the state spaces of the prototype programs grow faster. This is further confirmed in the right graph, where we see that the ratio between the state space sizes increases as $\square$ increases.

5.4 Tower of Hanoi

The tower of Hanoi is a classical example of a problem with an extremely large state space. There are three poles. One of the poles has $\square$ rings on it. Each ring is smaller than the one below it. The task is to move all rings, one at a time, to one other pole. At no point may a ring be placed on top of a smaller ring. Figure 5.6 provides an artistic impression of the problem. Here the $\square = 4$ rings are placed around pole $\square$ and must be moved to either pole $\square$ or $\square$. In this experiment we compare the state space sizes of Promela programs produced by our pipeline with reference solutions, for varying values of $\square$, just like our experiment with the dining philosophers problem.

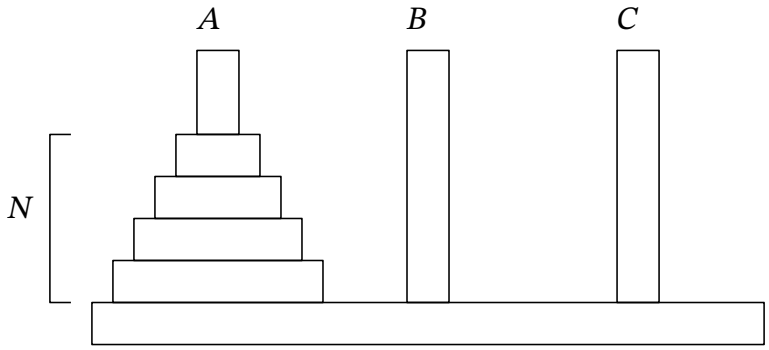

Figure 5.6 – The tower of Hanoi. The three poles are marked as □, □, and □. The amount of rings, □, is configurable. In this case we have □ = 4 rings.

The experiment setup follows the same structure as our experiment with the dining philosophers problem. We have two templates. One template produces AML models, and the other produces Promela programs. Both templates take □ as their only parameter. We call the Promela programs produced by our pipeline for the AML models "prototype programs", and we call the Promela programs produced by the other template "reference programs". Both kinds of programs follow the same structure. The goal of these programs is not to move all rings from one pole to another. Instead, these programs simply explore all possible states. Since it is always possible to move a ring from one pole to another, we will never encounter a deadlock. In Appendix C.4 we list both program kinds for □ = 3.

Table 5.3 shows the results for the reference and prototype programs. Just as we saw with the dining philosophers problem, we see that the prototype programs always have larger state spaces than their reference counterparts, and require a deeper search. However, there is also an important difference between the results of these two experiments. We observed a

Reference			Prototype		
□	States	Depth	□	States	Depth
3	28	40	3	706	329
4	82	112	4	1948	905
5	244	328	5	5674	2633
6	730	976	6	16852	7817
7	2188	2920	7	50386	23369
8	6562	8752	8	150988	70025
9	19684	26248	9	452794	209993
10	59050	78736	10	1358212	629897
11	177148	236200	11	4074466	1889609
12	531442	708592	12	12223228	5668745
13	1594324	2125768			
14	4782970	6377296			
15	14348908	19131880			

Table 5.3 – Comparing the state space sizes and the required search depth between the reference and prototype programs for the tower of Hanoi problem.

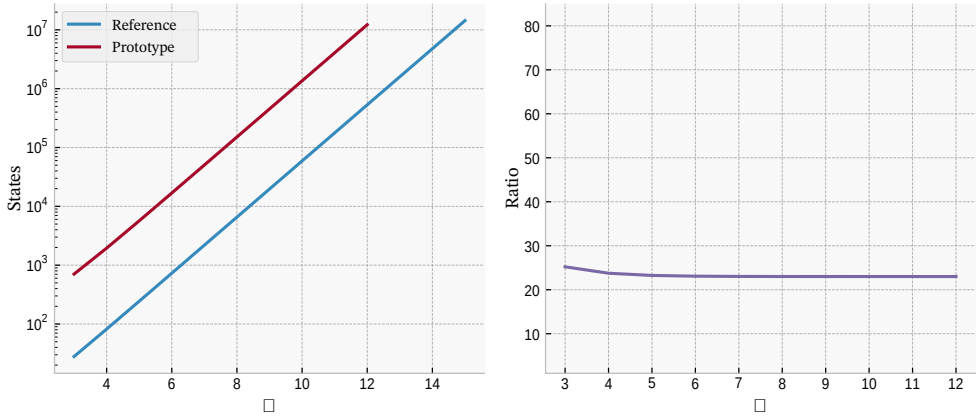

Figure 5.7 – Comparing the state space sizes between the reference and prototype programs. The graph on the left shows the growth for both programs on a logarithmic scale. The graph on the right shows the ratio between these two sizes.

difference in the growth rate between the prototype and reference programs for the dining philosophers problem; the state spaces of the prototype programs grew faster as □ increased. In contrast, we see similar growth rates between the two program kinds for the tower of Hanoi; the state space sizes of both the prototype and reference programs roughly triple for each increase of □. Figure 5.7 reflects this. In the left graph we see two near-parallel straight lines in logarithmic space, showing that both program kinds scale exponentially. While the prototype programs clearly contain more states, both program kinds grow at roughly the same rate. Consequently, the ratio between the prototype state space size and the reference prototype state space varies little as □ varies. This is reflected in the right graph.

This discrepancy in growth rate can be explained by inspecting the structure of these two problems. For the tower of Hanoi problem, a move consists of moving a ring from one pole to another. There are six possible moves, of which two are enabled at any time. This number of moves is independent of the number of rings. In contrast, the number of forks that can be picked up and put down does depend on the number of philosophers. For both problems, these actions are executed as part of a synchronization. Consequently, the number of flags we use in the prototype Hanoi programs is constant, whereas the number of flags we use in the dining philosopher prototype programs increases as the number of philosophers increases. Therefore, as we increase the number of philosophers the flags have a stronger effect on the overall state space size.

Chapter 6

Relation to Symbolic Transition Systems

In Chapter 1 we briefly discussed the relation between AML models and IOSTSs. In this chapter we elaborate on this relation, and discuss whether our model checking pipeline can also be applied to IOSTSs.

While AML contains structures tailored towards MBT that have no IOSTS counterpart, AML models and IOSTSs are structurally and semantically similar: both describe systems consisting of one or more sub-systems that exchange parameterized messages over transitions that may or may not be enabled depending on the message parameter values and the state of each synchronizing sub-system. In addition, the concept of being *open* or *closed* applies to both. Our pipeline contains two major steps; closing the open AML model, and translating the resulting closed model to Promela. We shall see that we can generalize our closure procedure from Chapter 3 to work for all IOSTSs. Translating a closed IOSTS to Promela using the mapping we defined in Chapter 4, however, only works for a particular class of IOSTS, namely pairwise-synchronizing IOSTSs. We discuss this translation in Section 6.1. We then discuss the generalizability of our closure procedure in Section 6.2.

6.1 IOSTSs Translation

Both AML processes and Promela processes synchronize in a pairwise fashion. The synchronization mechanism we use in the generated Promela programs is not able to facilitate the synchronization of more than two processes. For this sole reason, we can only apply this mapping to IOSTSs that also synchronize in a pairwise fashion. In this section, we discuss what IOSTSs are, how they synchronize, and how pairwise-synchronizing IOSTSs constitute a particular class of IOSTS.

There are multiple sources in the literature that provide an IOSTS definition, *e.g.*, [6] and [39]. We will use the definition from [4]. We prefer this definition as it is simpler, and IOSTSs are defined as a particular kind of STS, making the definition more generic. [6] and [39] do, however, define synchronization operators for IOSTSs, whereas [4] does not. However,

we find these synchronization operators too restrictive; they allow the synchronization of IOSTSs only under certain conditions. These conditions are often not met in AML models. Consequently, we will need to define our own synchronization operator for IOSTSs.

The definition of an IOSTS is a simple extension of the definition of an STS [4]. Let us start with the definition of the latter.

Definition 6.1 (Symbolic Transition System (STS))**.** A symbolic transition system [5] is a tuple $(\square, \square_0, \square, \square, \square, \Lambda, \rightarrow)$, where

- $\square$ is the set of *locations*,
- $\square_0 \in \square$ is the *initial location*,
- $\square$ is the set of *location variables*,
- $\square \in \square(\square)^{\square}$ is the *initialization* of the location variables. It is a function[1] $\square \rightarrow \square(\square)$, where $\square(\square)$ is the set of *ground terms* [48]. In general, $\square(\square)$ is the set of terms over the set of variables $\square$ [48]. In this case $\square$ assigns a term without variables to each location variable,
- $\square$ is the set of *interaction variables*. This set is disjoint from $\square$: $\square \cap \square = \square$,
- Λ is the set of *gates*. $\square \in \Lambda$ is the unobservable gate. We write $\Lambda_{\square}$ for $\Lambda \setminus \{\square\}$. Each gate $\square \in \Lambda$ has an associated arity, denoted as $\square\square\square\square(\square)$. Each gate also has a type, denoted as $\square\square\square\square(\square)$. The type of $\square$ is a tuple of length $\square\square\square\square(\square)$ of distinct interaction variables. The arity of the unobservable gate is 0,
- $\rightarrow \subseteq \square \times \Lambda_{\square} \times \square(\square \cup \square) \times \square(\square \cup \square)^{\square} \times \square$ is the *switch relation*. Instead of writing $(\square, \square, \square, \square, \square')$ $\in \rightarrow$, we write $\square \xrightarrow{\square,\square,\square} \square'$ $\in \rightarrow$ to denote the same thing. For any tuple $\square \xrightarrow{\square,\square,\square} \square'$ $\in \rightarrow$, $\square$ represents the switch condition and $\square$ represents the update mapping. $\square(\square \cup \square)$ is the set of first order formulas over $\square \cup \square$. We require $\square\square\square\square(\square) \subseteq \square\square\square\square(\square)$ $\cup$ $(\square \cup \square\square\square\square(\square))$, where the $\square\square\square\square(\square)$ function returns the set of free variables in expression $\square$, and $\square\square\square(\square)$ returns the variables used in update mapping $\square$, respectively.

In short, an STS describes a system. Its current state is determined by its current location, and the values for the location variables. Symbols $\square_0$ and $\square$ determine the initial location and initial location variable values, respectively. From its current location, the STS may advance to a next location by traversing over a switch. This switch has a gate $\square$ that determines the interaction variables used for this switch. If, given the current valuation for both $\square$ and $\square\square\square\square(\square)$, the switch condition is true, then the switch is *enabled*, otherwise it is *disabled*. We may only traverse over enabled switches. When multiple switches are enabled at the same time, the switch to traverse over is selected non-deterministically. Upon traversing over a switch, the update mapping assigns new values to the location variables. The unobservable gate is denoted $\square \in \Lambda$. All other gates are observable.

An IOSTS [4] is a special kind of STS. In such a system, the gates are partitioned into a set of *input* gates and a set of *output* gates, respectively denoted as $\Lambda_{\square}$ and $\Lambda_{\square}$.

[1] Let $\square$ and $\square$ be two sets. $\square^{\square}$ denotes the set of all possible functions $\square \rightarrow \square$.

The relation between AML processes and IOSTSs becomes clear when we relate the names of their respective elements. Locations are state labels, and location variables are process variables. A gate is a label. An input gate is a stimulus label, and an output gate is a response label. Labels and gates have types. The set of interaction variables corresponds to the union of the types of all labels; the set of all label parameters. A switch is a transition. Like a switch, a transition has a label, a guard, and an update function.

Now that we have an understanding of the semantics of IOSTSs, we can define what it means for IOSTSs to *synchronize* or *shake hands*. Say we have two IOSTSs. Both systems change state synchronously when one's output is the other's input. That is, both systems traverse over a switch with a gate that is an input gate in one system, and an output gate in the other. All switches with other gates only change the state of one particular system. Below, we provide our formal definition of IOSTS synchronization.

Definition 6.2 (Handshaking IOSTSs)**.** Let $\mathcal{T}_i = (L_i, l_{0,i}, V_i, \Theta_i, \Lambda_i, \Lambda_{I,i} \cup \Lambda_{O,i}, \rightarrow_i)$, for $i = 1, 2$ be two IOSTSs. We define the set of synchronization gates as:

$$\Gamma = \{\gamma \mid \gamma \in \Lambda_{I,1} \wedge \gamma \in \Lambda_{O,2}\} \cup \{\gamma \mid \gamma \in \Lambda_{O,1} \wedge \gamma \in \Lambda_{I,2}\}$$

The synchronization of $\mathcal{T}_1$ and $\mathcal{T}_2$ is defined as:

$$\mathcal{T}_1 \parallel_\Gamma \mathcal{T}_2 = (L_1 \times L_2, (l_{0,1}, l_{0,2}), (V_1, V_2), (\Theta_1, \Theta_2), \Lambda_1 \cup \Lambda_2, \rightarrow)$$

Note that $\Lambda_i = \Lambda_{I,i} \cup \Lambda_{O,i}$. If Γ is the entire set of shared gates, then we omit the subscript from the synchronization operator. We define the $\rightarrow$ relation in two parts:

- $\gamma \in \Gamma \cap (\Lambda_1 \cap \Lambda_2 \cap \{\gamma\})$:

$$\frac{l_1 \xrightarrow{\gamma,\pi,\varphi} l_1' \quad \varphi \rightarrow_1 \quad \theta_2 \in \Theta_2}{(l_1, l_2) \xrightarrow{\gamma,\pi,(\varphi,\varphi)} (l_1', l_2)} \quad \text{and} \quad \frac{l_2 \xrightarrow{\gamma,\pi,\varphi} l_2' \quad \varphi \rightarrow_2 \quad \theta_1 \in \Theta_1}{(l_1, l_2) \xrightarrow{\gamma,\pi,(\varphi,\varphi)} (l_1, l_2')}$$

- $\gamma \notin \Gamma$:

$$\frac{l_1 \xrightarrow{\gamma,\pi_1,\varphi_1} l_1' \quad \varphi_1 \rightarrow_1 \quad l_2 \xrightarrow{\gamma,\pi_2,\varphi_2} l_2' \quad \varphi_2 \rightarrow_2}{(l_1, l_2) \xrightarrow{\gamma,\pi_1 \cup \pi_2,(\varphi_1,\varphi_2)} (l_1', l_2')}$$

This definition and its notation takes inspiration from the definition of the synchronization operator for LTSs found in [49]. The locations of the synchronized system are pairs of locations of the original two IOSTSs. The location variables are placed in a tuple to avoid naming conflicts. Consequently, the initialization functions are placed in a tuple as well. The update mappings in the switch relation also form a pair, where the first mapping updates the location variables of the first system, and the second updates the location variables of the second system. Switches with gates outside of Γ only update the state of one sub-system, whereas switches with gates in Γ update the states of both sub-systems in a single step.

The synchronization operator we defined is binary; it produces the synchronization of two

IOSTSs. However, this does not mean that we cannot synchronize three or more systems. Say we have three IOSTSs, $\mathcal{T}_1$, $\mathcal{T}_2$, and $\mathcal{T}_3$. The synchronization operator is commutative. It is associative when the same set of synchronization gates is each between each operand:

$$(\mathcal{T}_1 \;|[\Gamma]|\; \mathcal{T}_2) \;|[\Gamma']|\; \mathcal{T}_3 = \mathcal{T}_1 \;|[\Gamma]|\; (\mathcal{T}_2 \;|[\Gamma']|\; \mathcal{T}_3) \quad \text{for } \Gamma = \Gamma' \cap (\Lambda_1 \cap \Lambda_2 \cap \Lambda_3).$$

Thus, we can synchronize $\mathcal{T}_1$, $\mathcal{T}_2$, and $\mathcal{T}_3$ by first synchronizing two systems, and synchronizing the result with the leftover system, yielding the synchronization of all three. Which systems we synchronize first may influence the final result. The operator we defined allows for *mulitway* synchronization; many sub-systems may take part in a handshake, upon which they all change state. Thus, two *or more* processes shake hands.

The Promela mapping we defined in Chapter 4 does not work for IOSTSs due to the two-step synchronization mechanism not being able to facilitate the synchronization of more than two processes. However, it is possible to synchronize IOSTSs in a pairwise fashion using a multiway synchronization operator. Let us assume we have three IOSTSs, $\mathcal{T}_1$, $\mathcal{T}_2$, and $\mathcal{T}_3$. We want to synchronize these systems as follows:

$$\mathcal{T} = (\mathcal{T}_1 \;|[\Gamma]|\; \mathcal{T}_2) \;|[\Gamma']|\; \mathcal{T}_3.$$

These systems synchronize pairwise if $\Gamma' \cap (\Lambda_1 \cap \Lambda_2) = \emptyset$. That is, Γ' must not contain a gate that is in both Λ_1 and Λ_2. By carefully renaming the gates of the systems taking part in the synchronization we can ensure that each process synchronizes pairwise. The multiway synchronization definition still allows us to make processes synchronize in a pairwise fashion. The other way around would have been impossible. The multiway definition is therefore simply more generally applicable. Our verification technique works for a specific class of IOSTSs, namely pairwise-synchronizing open IOSTSs.

6.2 Closing IOSTSs

In the previous section we argued that we can translate an IOSTS to Promela using the mapping defined in Chapter 4, as long as the sub-systems of the IOSTS synchronize in a pairwise fashion. This section assumed that the IOSTS is closed[2]. In this section we show what it means for an IOSTS to be open, and how the algorithm in Chapter 3 can be adapted to close such a system.

The literature is not devoid of publications that discuss open IOSTSs [39], [50]. However, in [50] IOSTSs may not communicate with each other. All communication occurs with the environment. In [39] no formal definition of open IOSTSs is provided. We provide our own definition of open IOSTSs that does allow inter-process communication.

An open IOSTS contains behavior that is dependent on the behavior of an external agent, the environment. Thus, we must determine which gates are used for communication with the environment. We define a subset of the input gates and a subset of the output gates that form exactly the set of gates used for communication with the environment. An IOSTS is

[2] Remember that the Promela translation step is performed on a closed AML model

closed when both these subsets are empty, otherwise it is open.

Definition 6.3 (Open IOSTS). Let $\mathcal{P} = (L, l_0, V, \iota, I, \Lambda_I \cup \Lambda_O, \rightarrow)$ be an IOSTS. Let $\Lambda_{I,o} \subseteq \Lambda_I$ and $\Lambda_{O,o} \subseteq \Lambda_O$ be the open input and output gates respectively. $\mathcal{P}$ is *closed* when $\Lambda_{I,o} = \Lambda_{O,o} = \emptyset$, otherwise it is *open*.

Say we have an open IOSTS, $\mathcal{P}_1$. We can close this system by synchronizing it with closed IOSTS $\mathcal{P}_2$, when the common set of gates, G, contains at least all open gates of $\mathcal{P}_1$:

$$\mathcal{P}_1 \parallel_G \mathcal{P}_2 \quad \text{where } (\Lambda_{I,o,1} \cup \Lambda_{O,o,1}) \subseteq G$$

A *minimal* environment process should only provide input to all open input transitions, while catching the output of all open output transitions. It has no other behavior. Thus;

$$\mathcal{P}_1 \parallel_G \mathcal{P}_2 \text{ where } G = \Lambda_2 = \Lambda_{I,o,1} \cup \Lambda_{O,o,1}$$

In Algorithm 3 we present a straightforward method for closing an IOSTS $\mathcal{P}$ by generating such a minimal environment process and synchronizing the two processes. It is essentially the same algorithm we used in Chapter 3, but now applied to an IOSTS. We create an environment process containing a single location. We call this the "home" location. Since this is the only location in the process, it is also the initial one. The environment process does not contain any location variables. Consequently, its initialization function is the empty function. Its gates are exactly the open gates of $\mathcal{P}$, but in the opposite direction: an input gates becomes an output gate, and an output gate becomes an input gate. For each of these gates we add a switch with the home location as both the source and target location. For each switch, the constraint is $\top$, and the update function is empty, as there are no location variables to update. The set of interaction variables is the union of the types of all open gates. This procedure generates a simple environment process with only a single location, "home", with self-loops that are always enabled. In the synchronized system the local state of

Algorithm 3 IOSTS closure

procedure CLOSE($\mathcal{P} = (L, l_0, V, \iota, I, (\Lambda_{I,c} \cup \Lambda_{I,o}) \cup (\Lambda_{O,c} \cup \Lambda_{O,o}), \rightarrow)$)
  $L_e := \{\text{home}\}$ ▷ e for "environment"
  $l_{0,e} := \text{home}$
  $V_e := \emptyset$
  $\iota_e := \emptyset$
  $I_e := \bigcup \{ x \in \mathit{type}(g) \mid g \in \Lambda_{I,o}, \Lambda_{O,o} \}$ ▷ We treat a tuple as a set here
  $\Lambda_e := \mathit{reverse}(\Lambda_{I,o} \cup \Lambda_{O,o})$ ▷ Swap the $\Lambda_{I,o}$ and $\Lambda_{O,o}$ partitions
  $\rightarrow_e := \emptyset$
  for $g \in \Lambda_e$ **do**
    $\rightarrow_e := \rightarrow_e \cup \{\text{home} \xrightarrow{g,\top,\emptyset} \text{home}\}$
  end for
  $\Lambda_{I,o} := \Lambda_{O,o} := \emptyset$
  return $\mathcal{P} \parallel_{\Lambda_e} (L_e, l_{0,e}, V_e, \iota_e, I_e, \Lambda_e, \rightarrow_e)$
end procedure

 is solely responsible for determining whether a particular transition of the environment process is enabled. For all open gates of there exists a single transition in the environment process, thus all open input transitions can receive input from the environment, and all open output transitions can freely send messages to the environment. We reassign the sets of open gates to the empty set, thus giving us a closed model.

Chapter 7

Discussion and Conclusion

We started this thesis by asking how we can verify safety properties of AML models. We made an early decision to use an existing model checker, SPIN, instead of building a new one. SPIN is a state-of-the-art explicit model checker capable of verifying safety properties. In addition, its input language, Promela, is simple. As work on the thesis progressed it became clear that translating AML to Promela was not as straightforward as anticipated. We decided to reduce the scope by focusing solely on the verification of deadlock-freedom of AML models. To this end we have designed, implemented, and experimented with our model checking pipeline. It is fully capable of verifying deadlock-freedom of AML models that lie within our scope, and reporting the steps that lead to deadlock states. To get to this point we have had to overcome several obstacles.

We consider the synchronization mechanism one of the largest obstacles. Because the constraints of receive transitions in AML models may refer to label parameters, we had to come up with a way to make Promela processes aware of these label parameter values when a synchronization attempt is made. Only then can the set of enabled transitions be determined. Taking inspiration from [41], we used a two-step synchronization mechanism. A process first receives a label accompanied by parameters, and then, depending on whether at least one if its outgoing transitions is enabled or not, it returns either an acknowledgment, or a non-acknowledgment. If all processes reject each message they can receive, and none can advance via a □-step, then the model is in a deadlock. However, as a consequence of synchronizations comprising multiple steps, and the first step being enabled solely depending on the sender's state, such a deadlock could occur as a livelock in the Promela program. This problem was noticed by [41], but remained unsolved. To turn these livelocks into deadlocks, we add flags to our Promela programs. With these flags we maintain a record of rejected label and parameter combinations. With this information processes are prevented from attempting synchronizations using messages that have been rejected previously. In Section 4.3 we found evidence that in certain models, these flags may enlarge the state space considerably. However, we also saw that for the BRP the overhead is much more manageable. Despite the addition of states that do not exist in the AML models, we provided evidence in Section 5.1 that we do indeed find *all* present deadlocks. No deadlocks were added, and none were missed.

The AML models we verify may be open. At the start of this thesis, we considered this problem our main obstacle. In hindsight, this was not a big issue at all. We have successfully devised an algorithm that closes AML models by generating an environment process and rerouting all external communication to it. The environment processes we generate provide all relevant inputs, while catching all outputs on previously external transitions. A similar closure procedure is used in [41]. Though, there is a difference. In [41] an environment process is generated *for each* process of the input model. We generate a single environment process. We do not expect this difference to matter when it comes to the correctness of our verification procedure, nor the number of states of the resulting Promela programs. However, to make our closure procedure function correctly, we had to rename the AML labels on external channels. This is not necessary when using the closure method from [41]. Had we realized this sooner, we would have used this method.

In our experiments we saw that the Promela programs we generate consistently contain more states than the reference models we compared them with. This raises questions about the synchronization mechanism, the addition of flags, and the mapping in general. Our mapping is by no means the only possible one. Other mappings may have particular benefits that would make it better than ours, at least in some situations. For example, mappings may utilize SPIN's C-extensions to hide certain information, resulting in a smaller state space, as we explained in Section 4.6. However, in Section 5.1 we also found evidence that the Promela programs we produce contain the exact same number of deadlocks as the AML input model. Thus, these superfluous states do not seem to affect the correctness of our results.

We have added support for a datatype that is not supported by SPIN: strings. We translate each AML string value to a unique integer. Because of our scoping constraints, we could easily extract each string value that could occur in any state. Thus, we have shown how we can still verify properties of models containing variables with types that SPIN does not support. These restrictions also allowed us to extract label parameters from AML models and transcribe them to Promela with relative ease. These restrictions are not realistic; they are often violated in real-life models. In real-life models these constraints often induce far greater numbers of parameter mappings than we have seen during this project. Often so many, that transcribing all these mappings to Promela is infeasible. If we were to lift these constraints, the analysis required to find all string values would be more complicated. In fact, such an analysis would most likely require the exploration of all states in the AML model. All states can be explored using a depth-first search algorithm, and adding support for deadlock detection to such an algorithm is trivial. Such an algorithm would form a solid foundation to further develop a model checker capable of verifying safety properties. When building a new model checker for verifying properties of AML models, the string and parameter extraction problems are no longer an issue. Strings no longer need to be translated, and label parameter values can be generated on the fly based on the state of the synchronizing processes and the constraints of their synchronizing transitions.

In conclusion, we have demonstrated that it is feasible to build a model checker that verifies deadlock-freedom of AML models using an existing model checker. We encountered several problems during this project. We hope that our solutions may serve as inspiration for future works. In our experiments, we found that our prototype does produce superfluous states, but

it does not report false results. Of course, these experiments do not *prove* that the mapping is correct. They do, however, provide confidence. We must, however, note once more that the restrictions we have placed on the AML models we verify are not realistic. We believe that we have demonstrated that model checking AML models a useful and feasible endeavor, but we recommend that future research is aimed at either modifying an existing model checker, or building a new model checker for verifying properties of AML models specifically.

Future Work

We previously mentioned that there is a difference between our closure method and the method from [41]. In [41] an environment process is generated for each process in the model. We generate only a single process. We do not expect this difference to have a significant effect on the state space of the resulting Promela program. However, a smarter environment process may reduce the number of flags that we need, possibly reducing the overall state space. Further research regarding this topic may yield results that are applicable to the verification of other types of systems as well.

There is still room to continue working on our prototype. Our research question stated that we wanted to figure out how to model check safety properties of AML models. We have subsequently reduced the scope to the verification of deadlock-freedom. The verification of every safety property can be reduced to a deadlock-freedom verification problem [51]. For this reason, we suspect that adding support for the verification of other safety properties would be a relatively straightforward task. Still, these properties themselves are often stated using a form of temporal logic. SPIN supports LTL [9] formulas in the form of so-called *never*-claims. A temporal logic specific to the domain of STSs does exist [39]. A potential avenue of research then, would be to investigate the feasibility of translating such formulas to never claims.

Bibliography

[1] J. Tretmans, "Model Based Testing with Labelled Transition Systems," in *Formal Methods and Testing: An Outcome of the FORTEST Network, Revised Selected Papers*, R. M. Hierons, J. P. Bowen, and M. Harman, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–38, ISBN: 978-3-540-78917-8. DOI: 10.1007/978-3-540-78917-8_1. [Online]. Available: https://doi.org/10.1007/978-3-540-78917-8_1.

[2] H. van der Bijl, "On changing models in Model-Based Testing," Centre for Telematics and Information Technology (CTIT) / University of Twente, Netherlands, May 12, 2011, ISBN: 978-90-365-3195-5. DOI: 10.3990/1.9789036531955.

[3] M. Müller-Olm, D. Schmidt, and B. Steffen, "Model-Checking," in *Static Analysis*, A. Cortesi and G. Filé, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 1999, pp. 330–354, ISBN: 978-3-540-48294-9. DOI: 10.1007/3-540-48294-6_22.

[4] L. Frantzen, J. Tretmans, and T. A. C. Willemse, "A Symbolic Framework for Model-Based Testing," in *Formal Approaches to Software Testing and Runtime Verification*, K. Havelund, M. Núñez, G. Roşu, and B. Wolff, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2006, pp. 40–54, ISBN: 978-3-540-49703-5. DOI: 10.1007/11940197_3.

[5] L. Frantzen, J. Tretmans, and T. A. C. Willemse, "Test Generation Based on Symbolic Specifications," in *Formal Approaches to Software Testing*, J. Grabowski and B. Nielsen, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2005, pp. 1–15, ISBN: 978-3-540-31848-4. DOI: 10.1007/978-3-540-31848-4_1.

[6] V. Rusu, L. du Bousquet, and T. Jéron, "An approach to symbolic test generation," in *Integrated Formal Methods*, W. Grieskamp, T. Santen, and B. Stoddart, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 338–357, ISBN: 978-3-540-40911-3.

[7] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Transactions on Software Engineering*, vol. SE-3, no. 2, pp. 125–143, Mar. 1977, ISSN: 1939-3520. DOI: 10.1109/TSE.1977.229904.

[8] G. Holzmann, "The Model Checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, May 1997, ISSN: 1939-3520. DOI: 10.1109/32.588521.

[9] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (Sfcs 1977)*, Oct. 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

[10] N. Sidorova and M. Steffen, "Synchronous Closing of Timed SDL Systems for Model Checking," in *Verification, Model Checking, and Abstract Interpretation*, A. Cortesi, Ed.,

ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2002, pp. 79–93, ISBN: 978-3-540-47813-3. DOI: 10.1007/3-540-47813-2_6.

[11] J. P. Queille and J. Sifakis, "Specification and verification of concurrent systems in CESAR," in *International Symposium on Programming*, M. Dezani-Ciancaglini and U. Montanari, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 337–351, ISBN: 978-3-540-39184-5.

[12] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Logics of Programs*, D. Kozen, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 52–71, ISBN: 978-3-540-39047-3.

[13] W. Visser, M. B. Dwyer, and M. Whalen, "The hidden models of model checking," *Software & Systems Modeling*, vol. 11, no. 4, pp. 541–555, Oct. 1, 2012, ISSN: 1619-1374. DOI: 10.1007/s10270-012-0281-9. [Online]. Available: https://doi.org/10.1007/s10270-012-0281-9 (visited on 02/04/2021).

[14] E. Emerson and J. Y. Halpern, "Decision procedures and expressiveness in the temporal logic of branching time," *Journal of Computer and System Sciences*, vol. 30, no. 1, pp. 1–24, Feb. 1985, ISSN: 00220000. DOI: 10.1016/0022-0000(85)90001-7. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/0022000085900017 (visited on 08/16/2021).

[15] M. Y. Vardi and P. Wolper, "Automata-theoretic techniques for modal logics of programs," *Journal of Computer and System Sciences*, vol. 32, no. 2, pp. 183–221, Apr. 1, 1986, ISSN: 0022-0000. DOI: 10.1016/0022-0000(86)90026-7. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0022000086900267 (visited on 06/07/2021).

[16] A. Pnueli and A. Zaks, "On the Merits of Temporal Testers," in *25 Years of Model Checking: History, Achievements, Perspectives*, ser. Lecture Notes in Computer Science, O. Grumberg and H. Veith, Eds., Berlin, Heidelberg: Springer, 2008, pp. 172–195, ISBN: 978-3-540-69850-0. DOI: 10.1007/978-3-540-69850-0_11. [Online]. Available: https://doi.org/10.1007/978-3-540-69850-0_11 (visited on 01/22/2021).

[17] W. Visser and H. Barringer, "Practical CTL* model checking: Should SPIN be extended?" *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 350–365, Mar. 1, 2000, ISSN: 1433-2779. DOI: 10.1007/s100090050042. [Online]. Available: https://doi.org/10.1007/s100090050042 (visited on 06/18/2021).

[18] L. Lamport, ""Sometime" is sometimes "not never": On the temporal logic of programs," in *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '80, New York, NY, USA: Association for Computing Machinery, Jan. 28, 1980, pp. 174–185, ISBN: 978-0-89791-011-8. DOI: 10.1145/567446.567463. [Online]. Available: https://doi.org/10.1145/567446.567463 (visited on 08/12/2021).

[19] E. A. Emerson and J. Y. Halpern, ""Sometimes" and "not never" revisited: On branching versus linear time temporal logic," *Journal of the ACM*, vol. 33, no. 1, pp. 151–178, Jan. 2, 1986, ISSN: 0004-5411. DOI: 10.1145/4904.4999. [Online]. Available: https://doi.org/10.1145/4904.4999 (visited on 05/31/2021).

[20] R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran, "MOCHA: Modularity in model checking," in *Computer Aided Veri cation*, A. J. Hu

and M. Y. Vardi, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 1998, pp. 521–525, ISBN: 978-3-540-69339-0. DOI: 10.1007/BFb0028774.

[21] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Progress on the state explosion problem in model checking," in *Informatics: 10 Years Back, 10 Years Ahead*, R. Wilhelm, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 176–194, ISBN: 978-3-540-44577-7. DOI: 10.1007/3-540-44577-3_12. [Online]. Available: https://doi.org/10.1007/3-540-44577-3_12.

[22] K. L. McMillan, "Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits," in *Computer Aided Verification*, G. von Bochmann and D. K. Probst, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 164–177, ISBN: 978-3-540-47572-9.

[23] J. Esparza, "Model checking using net unfoldings," *Science of Computer Programming*, vol. 23, no. 2-3, pp. 151–195, Dec. 1994, ISSN: 01676423. DOI: 10.1016/0167-6423(94)00019-0. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/0167642394000190 (visited on 05/27/2021).

[24] J. Esparza, S. Römer, and W. Vogler, "An Improvement of McMillan's Unfolding Algorithm," *Formal Methods in System Design*, vol. 20, no. 3, pp. 285–310, May 1, 2002, ISSN: 1572-8102. DOI: 10.1023/A:1014746130920. [Online]. Available: https://doi.org/10.1023/A:1014746130920 (visited on 05/30/2021).

[25] D. Peled, "Partial-order reduction," in *Handbook of Model Checking*, E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds., Cham: Springer International Publishing, 2018, pp. 173–190, ISBN: 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_6. [Online]. Available: https://doi.org/10.1007/978-3-319-10575-8_6.

[26] J. Esparza and K. Heljanko, *Unfoldings: A Partial-Order Approach to Model Checking*, ser. Monographs in Theoretical Computer Science An EATCS Series. Berlin, Heidelberg: Springer-Verlag Berlin Heidelberg, 2008, ISBN: 978-3-540-77426-6.

[27] S. Graf and H. Saidi, "Construction of abstract state graphs with PVS," in *Computer Aided Verification*, O. Grumberg, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 72–83, ISBN: 978-3-540-69195-2.

[28] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani, "Automatic predicate abstraction of C programs," in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, ser. PLDI '01, New York, NY, USA: Association for Computing Machinery, May 1, 2001, pp. 203–213, ISBN: 978-1-58113-414-8. DOI: 10.1145/378795.378846. [Online]. Available: https://doi.org/10.1145/378795.378846 (visited on 06/06/2021).

[29] J. Esparza, "Verification of Systems with an Infinite State Space," in *Modeling and Verification of Parallel Processes: 4th Summer School, MOVEP 2000 Nantes, France, June 19–23, 2000 Revised Tutorial Lectures*, ser. Lecture Notes in Computer Science, F. Cassez, C. Jard, B. Rozoy, and M. D. Ryan, Eds., Berlin, Heidelberg: Springer, 2001, pp. 183–186, ISBN: 978-3-540-45510-3. DOI: 10.1007/3-540-45510-8_8. [Online]. Available: https://doi.org/10.1007/3-540-45510-8_8 (visited on 02/04/2021).

[30] A. W. Lin, "Model Checking Infinite-State Systems: Generic and Specific Approaches," University of Edinburgh, Edinburgh, 2010, 294 pp.

[31] R. Pelánek, "Fighting State Space Explosion: Review and Evaluation," in *Formal Methods for Industrial Critical Systems*, D. Cofer and A. Fantechi, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2009, pp. 37–52, ISBN: 978-3-642-03240-0. DOI: 10.1007/978-3-642-03240-0_7.

[32] D. Harel and A. Pnueli, "On the Development of Reactive Systems," in *Logics and Models of Concurrent Systems*, K. R. Apt, Ed., ser. NATO ASI Series, Berlin, Heidelberg: Springer, 1985, pp. 477–498, ISBN: 978-3-642-82453-1. DOI: 10.1007/978-3-642-82453-1_17.

[33] N. Sidorova and M. Steffen, "Embedding Chaos," in *Static Analysis*, P. Cousot, Ed., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2001, pp. 319–334, ISBN: 978-3-540-47764-8. DOI: 10.1007/3-540-47764-0_18.

[34] M. B. Dwyer and C. S. Pasareanu, "Filter-based model checking of partial systems," *ACM SIGSOFT Software Engineering Notes*, vol. 23, no. 6, pp. 189–202, Nov. 1, 1998, ISSN: 0163-5948. DOI: 10.1145/291252.288307. [Online]. Available: http://doi.org/10.1145/291252.288307 (visited on 02/05/2021).

[35] O. Tkachuk, M. B. Dwyer, and C. S. Pasareanu, "Automated environment generation for software model checking," in *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, Oct. 2003, pp. 116–127. DOI: 10.1109/ASE.2003.1240300.

[36] N. Sidorova and M. Steffen, "Verifying Large SDL-Specifications Using Model Checking," in *SDL 2001: Meeting UML*, R. Reed and J. Reed, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2001, pp. 403–420, ISBN: 978-3-540-48213-0. DOI: 10.1007/3-540-48213-X_25.

[37] J. Stoel, T. van der Storm, and J. Vinju, "Modeling with Mocking," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, Apr. 2021, pp. 59–70. DOI: 10.1109/ICST49551.2021.00018.

[38] C. Gaston, P. Le Gall, N. Rapin, and A. Touil, "Symbolic execution techniques for test purpose definition," in *Testing of Communicating Systems*, M. Ü. Uyar, A. Y. Duale, and M. A. Fecko, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–18, ISBN: 978-3-540-34185-7.

[39] M. Aiguier, C. Gaston, P. Le Gall, D. Longuet, and A. Touil, "A temporal logic for input output symbolic transition systems," in *12th Asia-Pacific Software Engineering Conference (APSEC'05)*, 2005, p. 8. DOI: 10.1109/APSEC.2005.19.

[40] C. Rütz, "Model Checking of Symbolic Transition Systems with SMT Solvers," M.S. thesis, Radboud University, Nijmegen, Nov. 8, 2013, 29 pp. [Online]. Available: https://www.axini.com/files/2013_carsten_ruetz.pdf (visited on 05/16/2021).

[41] K. de Berk, "Model Checking Test Models," University of Amsterdam, Amsterdam, 2014, 69 pp. [Online]. Available: https://www.axini.com/files/2014_kevin_de_berk.pdf (visited on 01/14/2021).

[42] Z. Baranová, J. Barnat, K. Kejstová, T. Kučera, H. Lauko, J. Mrázek, P. Ročkai, and V. Štill, "Model checking of c and C++ with DIVINE 4," in *Automated Technology for Verification and Analysis*, D. D'Souza and K. Narayan Kumar, Eds., Cham: Springer International Publishing, 2017, pp. 201–207, ISBN: 978-3-319-68167-2.

[43] G. Kant, A. Laarman, J. Meijer, J. van de Pol, S. Blom, and T. van Dijk, "LTSmin: High-performance language-independent model checking," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. Baier and C. Tinelli, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 692–707, ISBN: 978-3-662-46681-0.

[44] G. J. Holzmann and R. Joshi, "Model-driven software verification," in *Model Checking Software*, S. Graf and L. Mounier, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 76–91, ISBN: 978-3-540-24732-6.

[45] P. R. D'Argenio, J. .-P. Katoen, T. C. Ruys, and J. Tretmans, "The bounded retransmission protocol must be on time!" In *Tools and Algorithms for the Construction and Analysis of Systems*, E. Brinksma, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 416–431, ISBN: 978-3-540-68519-7.

[46] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi, "UPPAAL — a tool suite for automatic verification of real-time systems," in *Hybrid Systems III*, R. Alur, T. A. Henzinger, and E. D. Sontag, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 1996, pp. 232–243, ISBN: 978-3-540-68334-6. DOI: 10.1007/BFb0020949.

[47] C. A. R. Hoare, *Communicating Sequential Processes*, ser. Prentice-Hall International Series in Computer Science. Englewood Cliffs, N.J: Prentice/Hall International, 1985, 256 pp., ISBN: 978-0-13-153271-7.

[48] W. Fokkink, *Introduction to Process Algebra*, red. by W. Brauer, G. Rozenberg, and A. Salomaa, ser. Texts in Theoretical Computer Science. An EATCS Series. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, ISBN: 978-3-662-04293-9. DOI: 10.1007/978-3-662-04293-9. [Online]. Available: http://link.springer.com/10.1007/978-3-662-04293-9 (visited on 11/04/2020).

[49] C. Baier and J.-P. Katoen, *Principles of Model Checking*. Cambridge, Mass: The MIT Press, 2008, 975 pp., ISBN: 978-0-262-02649-9.

[50] N. Rapin, "Symbolic execution based model checking of open systems with unbounded variables," in *Tests and Proofs*, C. Dubois, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 137–152, ISBN: 978-3-642-02949-3.

[51] P. Godefroid and P. Wolper, "Using partial orders for the efficient verification of deadlock freedom and safety properties," in *Computer Aided Veri cation*, K. G. Larsen and A. Skou, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 1992, pp. 332–342, ISBN: 978-3-540-46763-2. DOI: 10.1007/3-540-55179-4_32.

[52] R. Gerth. (Jun. 1997). "Concise Promela Reference," [Online]. Available: http://spinroot.com/spin/Man/Quick.html (visited on 11/08/2021).

[53] G. J. Holzmann, *The Spin Model Checker: Primer and Reference Manual*, 4th print. Boston, Mass. Munich: Addison-Wesley, 2008, 596 pp., ISBN: 978-0-321-22862-8.

Glossary

AML Axini Modeling Language. 1, 4–7, 9–13, 16–19, 22–27, 29–32, 34–36, 38, 40–43, 45, 47–50, 52, 54–57, 60–62, 69–71, 74–77, 85, 87, 91, 96, 98, 99

BRP Bounded Retransmission Protocol. 46, 47, 49, 60, 87, 91, 96

DSL Domain Specific Language. 4

IOSTS Input-Output Symbolic Transition System. 1, 4, 5, 43–46, 54–58

LTS Labeled Transition System. 9, 56

MBT Model Based Testing. 1, 4, 5, 9, 54, 69, 70

STS Symbolic Transition System. 1, 4, 6, 9, 22, 54, 55, 62, 69

SUT System Under Test. 1, 4, 5, 9, 70

Appendix A

Language Atoms

A major part of the prototype involves the translation of an STS described in AML syntax to a Promela program. In this section we provide a short overview of both the Axini Modeling Language (AML) and Promela.

A.1 The Axini Modeling Language

The Axini Modeling Language (AML) is a domain-specific language used to describe AML models for the purpose of Model Based Testing (MBT). Listing A.1 shows an example of an AML model. We will use this example to illustrate AML constructs.

On the first two lines we define three *channels*, an external channel called `ext`, and two internal ones called `int1` and `int2`. An AML model consists of one or more *processes*. When the model is used for MBT, a testing environment is responsible for providing input and catching output on the external channels. The internal channels are used for inter-process communication.

We define two processes, `A` and `B`. Both process definitions start with a channel-block that defines the *stimuli* and *responses* the process may receive and send on a particular channel. Let us focus on process `A` for now. Process `A` states that it can receive stimuli with one of two *labels* on the external channel: `set` and `success`. Both labels come with a *label parameter*. The label parameter for the `set` label is `x`, and the label parameter for the `success` label is `new_i`. Both are of type integer. At times, we will refer to a label accompanied by label parameter values as a *message*. A label may only be defined in a single direction on a specific channel. For example, process `A` defines a `a` stimulus on channel `int1`. It is not allowed to also define `a` as a response on the same channel. For every label on an internal channel there must be one process that defines it as a stimulus, and one process that defines it as a response.

After the channel-blocks we define and initialize the *process variables*, `i` and `reached_home_by`. After that we define a *state label* using the `state` keyword. The first specified state label is the initial state label of the process. The *state* of a process is determined by its current state label and the values for its process variables. The state of the model is the combined states

of all processes. A state label may have zero or more outgoing transitions. A choice-block is used when there is more than one. Inside this choice-block an o-block defines a single transition. There are three kinds of transitions: 1) sending a message, 2) receiving a message, and 3) □-steps. The first two kinds exchange a message either with another process or the environment. A □-step defines an internal state change; the process changes its state without communicating. These steps are sometimes called *unobservable*. A transition is either *enabled* or *disabled*. Only enabled transitions may be executed. When there are multiple enabled transitions, the next transition to executed is non-deterministically chosen. Each transition may have a constraint expression and an update expression associated with it. When the transition is executed, its update expression assigns new values to the process variables. For send and receive transitions, this expression may refer to both process variables and the label parameters. For □-steps, the expression may only refer to process variables. Users may also define functions to use in both the constraint and update expressions. We consider these out of scope for this thesis. The constraint expression is a boolean expression. An empty constraint is always true. Like with the update expressions, the constraints of send transitions and receive transitions may refer to both the process variables and the message's label parameters. The constraints of □-steps may only refer to the process variables. A □-step is enabled if its constraint is satisfied. Send and receive transitions are enabled if their constraints are satisfied, *and* another process is in a state where it can exchange this message. For example, say process `A` is at state label `waiting` with $\square_{\square} = 4$, and process `B` is at state label `comm` with $\square_{\square} = 2$[1]. `A` cannot send `k(i)` (line 26) to process `B`, because, while the constraint of `A`'s transition is satisfied, the constraint of `B`'s (line 60) is not. If $\square_{\square} = 2$, then these transitions would be enabled. We end each transition with a `goto` statement, that states which state label to go to when the transition is executed. This `goto`-statement is optional. If it is left out, then the AML parser will create an anonymous state to advance to. At the top of the model we find a `timeout` statement. This statement is not relevant in the context of this thesis. It is only used for MBT to define a time limit on responses from the SUT. We added this statement for syntactical correctness.

A.2 Promela

Promela, or the Process Meta Language, is the input language for the SPIN model checker. SPIN takes a Promela program as input, and produces C code for a verifier specific to that program. The verifier can then be compiled and ran.

Listing A.2 shows part of the Promela program produced by our prototype for the AML model of Listing A.1. You can find the complete program on Page 80. Only parts of processes `A` and `B` are shown for the sake of brevity. The environment is not shown at all. We will not go into much detail here about the behavior of this program. We only want to introduce the syntax and semantics of Promela.

A Promela program consists of *processes* communicating over *channels*. We find two arrays of channels in this fragment, `chan_set_A` and `chan_k`. Both are declared globally. A channel has a *type*. All messages sent over a channel must have this type. A special type is the `mtype`

[1] This is not a reachable state in the model. This imaginary state serves merely as an illustration

```
internal 'int1', 'int2'
external 'ext'
timeout 1

process('A') {
  channel('int1') {
    stimulus 'a'
    response 'k', { 'x' => :integer }
  }
  channel('int2') {
    stimulus 'a'
  }
  channel('ext') {
    stimulus 'set', { 'x' => :integer }
    response 'success', { 'new_i' => :integer }
  }
  var 'i', :integer, 0
  var 'reached_home_by', :string, "init"

  state 'home'
  receive 'a', on: 'int1'
  goto 'waiting'

  state 'waiting'
  choice {
    o { receive 'set', constraint: '(x == 1 || x == 2) && i == 1', update: 'i = □  x + 1'
        send 'success', constraint: 'new_i == i'
        goto 'set' }
    o { send 'k', constraint: 'x == i', update: 'i = i + 1; reached_home_by = "□ k"'
        goto 'home' }
  }
  state 'set'
  choice {
    o { receive 'a', on: 'int2', update: 'i = 1; reached_home_by = "a"'
        goto 'home' }
    o { tau constraint: 'i == 0', update: 'reached_home_by = "tau"'
        goto 'home' }
  }
}

process('B') {
  channel('int1') {
    response 'a'
    stimulus 'k', { 'x' => :integer }
  }
  channel('int2') {
    response 'a'
  }
  channel('ext') {
    stimulus 'set', { 'x' => :integer }
  }
  var 'i', :integer, 0

  state 'home'
  send 'a', on: 'int1'
  goto 'comm'

  state 'comm'
  choice {
    o { receive 'k', constraint: 'x == i && i < 3', update: 'i = i + 1'
        goto 'home' }
    o { send 'a', on: 'int2'
        goto 'home' }
    o { receive 'set', constraint: '(x == 0 && i == 0) || (x == 1 && i == 0)'
        goto 'comm' }
  }
}
```

Listing A.1 – AML example model.

type. Programs may define up to 256 symbols that may be used as these "message types". These constants are comparable to enumerated types of C-like languages. All channels in this fragment exchange messages consisting of an `mtype` and an `int`. All channels have a capacity of 0, making them *rendezvous* channels. *Variable*s may be declared on a global scope, or they may be local to a process. In the fragment we find a globally declared `flag` integer variable, as well as two arrays, `flags_k` and `flags_set_A`. Process `A` has three local variables, `i`, `reached_home_by`, and `set_A_x`. Statements are evaluated top to bottom. A statement is either *blocking* or *enabled*. Simple statements, like assignments, never block. In contrast, a boolean expression may block. The expression `false` always blocks, while the expression `true` never blocks. Whether the expression `x + y == 3` blocks depends on the values of `x` and `y`.

Let us focus on process `A` for now. On line 11 we find a *label* called `label_A_waiting`. We can jump to this label using *goto*-statements. On the next line we find an *if*-statement. Such a statement defines a non-deterministic choice between one or more *option*s. The first option we find on line 13, and the second one on line 30. Whether an option is enabled or not depends on its first statement. If this first statement is enabled, then the option as a whole is enabled. This first statement is called the options' *guard*. If all guards block, then the if-statement as a whole is blocked. A special `else`-guard may be added. This guard is only enabled if none of the other guards are. We see that each option is wrapped by an `atomic` embrace. Like the options, an atomic block is enabled if its first statement is enabled. The first statement of the atomic on line 13 states the reception of a message, whereas the first statement of the atomic on line 30 states the dispatch of a message, denoted by question and exclamation marks respectively. When communicating over rendezvous channels, a process can send a message only when there is another process in state where it can receive this message. For example, on line 30 process `A` wants to send a `k(i)` message on channel `chan_k[0]`. This statement is only enabled if another process can receive this message. On line 52 we find that process `B` can receive these messages. Thus, the options on line 30 and line 52 are enabled simultaneously when process `A`'s current state is at the if-statement of line 12, and process `B`'s current state is at the if-statement of line 51. Such a message exchange is known as a *rendezvous*. However, that is not the complete story. On line 30 we find a conditional expression between the square brackets. This expression is used to pick either `chan_k[0]` or `chan_k[1]`. If the condition is false, then `chan_k[1]` is picked. We see that process `B` only listens on `chan_k[0]`. Thus, if process `A` attempts to send a message on `chan_k[1]`, then there is no process that listens for messages on this channel, and thus the rendezvous cannot occur.

We do not wrap each option in an atomic embrace for no reason. When a process enters an atomic block it becomes the sole process that is allowed to execute statements. If a statement in the atomic sequence blocks, then atomicity is lost. The progress may regain atomicity when the blocked statement becomes enabled again, after which the remainder of atomic block is executed. When a process executes a goto-statement in an atomic block it loses atomicity, unless it jumps into another atomic block. A process can transfer atomicity to another process by sending a message on a rendezvous channel while in atomic block, and the receiver's receive statement is also in an atomic block.

Let us assume that process `A` and `B` exchange a `k(i)` message on lines 30 and 52 respectively. We, again, focus solely on process `A`. On line 31 process `A` waits for an acknowledgment message from `B`. Once it receives this acknowledgment it enters a *deterministic step*, or `d_step`. Like in an atomic block, a `d_step` is enabled if its first statement is. As the name suggests, all statements inside the block must be deterministic. These statements are executed as if they are a single step. This reduces the number of states of the program. In each `d_step` in the program, we only find statements that cannot block. A blocking statement in a `d_step` is considered an error. Note that certain statements, like send, receive and goto-statements are not allowed in `d_steps`.

We have attempted to keep this description of Promela brief and focused on the parts we need for this thesis. For a more in-depth, but still concise description of the language, we refer the reader to the Concise Promela Reference[52]. For more information on SPIN and Promela, we refer the reader to [53].

```
mtype {a, set_A, k, success_A, a_0, set_B, ACK, NACK};
chan chan_set_A[2] = [0] of {mtype, int};
chan chan_k[2] = [0] of {mtype, int};
int flag = 0;
bit flags_k[1] = 0;
bit flags_set_A[2] = 0;
proctype A() {
  int i = 0;
  int reached_home_by = 4;
  int set_A_x = 0;
label_A_waiting:
  if
  :: atomic { chan_set_A[0] ? set_A(set_A_x) -> if
    :: (set_A_x == 1 || set_A_x == 2) && i == 1 -> {
      d_step {
        for (flag in flags_a_0) { flags_a_0[flag] = 0; }
        for (flag in flags_k) { flags_k[flag] = 0; }
        flag = 0;
        printf("A[label_A_waiting]: chan_set_A ? set_A(%d) -> label_A__2\n",
            □ set_A_x);
        i = set_A_x + 1;
        printf("A: [i = %d, reached_home_by = %d]\n", i, reached_home_by);
        set_A_x = 0;
      }
      sync ! ACK;
      goto label_A__2;
    }
    :: else -> sync ! NACK; set_A_x = 0; goto label_A_waiting;
    fi
  }
  :: atomic { chan_k[(!flags_k[0] && (i == i) -> 0 : 1)] ! k(i) -> if
    :: sync ? ACK -> {
      d_step {
        for (flag in flags_set_A) { flags_set_A[flag] = 0; }
        flag = 0;
        printf("A[label_A_waiting]: chan_k ! k(%d) -> label_A_home\n", i);
        i = i + 1; reached_home_by = 1;
        printf("A: [i = %d, reached_home_by = %d]\n", i, reached_home_by);
      }
      goto label_A_home;
    }
    :: sync ? NACK -> flags_k[0] = 1; goto label_A_waiting;
    fi
  }
  fi
}
proctype B() {
  int i_0 = 0;
  int k_x_0 = 0;
  int set_B_x = 0;
label_B_comm:
  if
  :: atomic { chan_k[0] ? k(k_x_0) -> if
    :: k_x_0 == i_0 && i_0 < 3 -> {
      d_step {
        for (flag in flags_k) { flags_k[flag] = 0; }
        flag = 0;
        printf("B[label_B_comm]: chan_k ? k(%d) -> label_B_home\n", k_x_0);
        i_0 = i_0 + 1;
        printf("B: [i_0 = %d]\n", i_0);
        k_x_0 = 0;
      }
      sync ! ACK;
      goto label_B_home;
    }
    :: else -> sync ! NACK; k_x_0 = 0; goto label_B_comm;
    fi
  }
}
```

Listing A.2 – Fragment of the Promela code produced for the AML model in Listing A.1.

Appendix B

Example Translation

In this thesis we have illustrated the steps of our pipeline using an example introduced in Section 2.1. As noted in this section, the example model we used is a toy model. It only serves to illustrate our mapping. It contains all the constructs that could make this translation difficult. In this chapter we provide the full code produced by our pipeline for each step of our pipeline. For convenience, we first reproduce the AML code for the example. We then show the closed model. We conclude this chapter with the code produced by the Promela translation procedure. First we show the naive translation, and then the translation actually produced by our pipeline. Let us start with the example model itself.

```
internal 'int1', 'int2'
external 'ext'
timeout 1

process('A') {
  channel('int1') {
    stimulus 'a'
    response 'k', { 'x' => :integer }
  }
  channel('int2') {
    stimulus 'a'
  }
  channel('ext') {
    stimulus 'set', { 'x' => :integer }
    response 'success', { 'new_i' => :integer }
  }
  var 'i', :integer, 0
  var 'reached_home_by', :string, "init"

  state 'home'
  receive 'a', on: 'int1'
  goto 'waiting'

  state 'waiting'
  choice {
    o { receive 'set', constraint: '(x == 1 || x == 2) && i == 1', update: 'i =
        □  x + 1'
        send 'success', constraint: 'new_i == i'
        goto 'set' }
    o { send 'k', constraint: 'x == i', update: 'i = i + 1; reached_home_by = "
        □ k"'
        goto 'home' }
  }
  state 'set'
```

```
  choice {
    o { receive 'a', on: 'int2', update: 'i = 1; reached_home_by = "a"'
        goto 'home' }
    o { tau constraint: 'i == 0', update: 'reached_home_by = "tau"'
        goto 'home' }
  }
}

process('B') {
  channel('int1') {
    response 'a'
    stimulus 'k', { 'x' => :integer }
  }
  channel('int2') {
    response 'a'
  }
  channel('ext') {
    stimulus 'set', { 'x' => :integer }
  }
  var 'i', :integer, 0

  state 'home'
  send 'a', on: 'int1'
  goto 'comm'

  state 'comm'
  choice {
    o { receive 'k', constraint: 'x == i && i < 3', update: 'i = i + 1'
        goto 'home' }
    o { send 'a', on: 'int2'
        goto 'home' }
    o { receive 'set', constraint: '(x == 0 && i == 0) || (x == 1 && i == 0)'
        goto 'comm' }
  }
}
```

Listing B.1 – AML example model.

The first step of the pipeline is parsing this model. We then have a data object representing the input model. The next step, the closing of the model, is performed on this object. The code below is not produced at any step of the pipeline. It is merely an illustration of the closed model resulting from the closure step performed on the input model.

```
internal 'int1', 'int2', 'internal_env'
timeout 1.0
process('A') {
  channel('int1') {
    stimulus 'a'
    response 'k', { 'x' => :integer }
  }
  channel('internal_env') {
    stimulus 'set_A', { 'x' => :integer }
    response 'success_A', { 'new_i' => :integer }
  }
  channel('int2') {
    stimulus 'a'
  }
  var 'i', :integer, 0
  var 'reached_home_by', :string, "init"

  state 'home'
  receive 'a', on: 'int1'
  goto 'waiting'

  state 'waiting'
  choice {
    o { receive 'set_A', constraint: '(x == 1 || x == 2) && i == 1', update: 'i
```

```
        □  = x + 1'
      goto 'gen_2' }
    o { send 'k', constraint: 'x == i', update: 'i = i + 1; reached_home_by = "
        □ k"'
      goto 'home' }
  }

  state 'gen_2'
  send 'success_A', constraint: 'new_i == i'
  goto 'set'

  state 'set'
  choice {
    o { receive 'a', on: 'int2', update: 'i = 1; reached_home_by = "a"'
      goto 'home' }
    o { tau constraint: 'i == 0', update: 'reached_home_by = "tau"'
      goto 'home' }
  }
}

process('B') {
  channel('int1') {
    response 'a'
    stimulus 'k', { 'x' => :integer }
  }
  channel('int2') {
    response 'a'
  }
  channel('internal_env') {
    stimulus 'set_B', { 'x' => :integer }
  }
  var 'i', :integer, 0

  state 'home'
  send 'a', on: 'int1'
  goto 'comm'

  state 'comm'
  choice {
    o { receive 'k', constraint: 'x == i && i < 3', update: 'i = i + 1'
      goto 'home'}
    o { send 'a', on: 'int2'
      goto 'home' }
    o { receive 'set_B', constraint: '(x == 0 && i == 0) || (x == 1 && i == 0)'
      goto 'comm' }
  }
}

process('closure_env') {
  channel('internal_env') {
    response 'set_A', { 'x' => :integer }
    stimulus 'success_A', { 'new_i' => :integer }
    response 'set_B', { 'x' => :integer }
  }

  state 'home'
  choice {
    o { send 'set_A'
      goto 'home' }
    o { receive 'success_A'
      goto 'home' }
    o { send 'set_B'
      goto 'home' }
  }
}
```

Listing B.2 – Closed version of the AML example model.

The closed model is the input model for the Promela translation step. With the naive

translation we described in Section 4.1 we get the following model.

```
mtype {a, a_0, k, set_A, set_B, success_A, ACK, NACK};

chan chan_a[2] = [0] of {mtype};
chan chan_a_0[2] = [0] of {mtype};
chan chan_k[2] = [0] of {mtype, int };
chan chan_set_A[2] = [0] of {mtype, int };
chan chan_set_B[2] = [0] of {mtype, int };
chan chan_success_A[2] = [0] of {mtype, int };
chan sync = [0] of {mtype};

proctype A() {
  int i = 0;
  int reached_home_by = 4;
  int set_A_x = 0;
label_A_home:
  if
  :: atomic { chan_a[0] ? a -> if
    :: true -> {
      sync ! ACK;
      goto label_A_waiting;
    }
    :: else -> sync ! NACK; skip; goto label_A_home;
    fi
  }
  fi
label_A_waiting:
  if
  :: atomic { chan_set_A[0] ? set_A(set_A_x) -> if
    :: (set_A_x == 1 || set_A_x == 2) && i == 1 -> {
      d_step {
        i = set_A_x + 1;
        set_A_x = 0;
      }
      sync ! ACK;
      goto label_A__2;
    }
    :: else -> sync ! NACK; set_A_x = 0; goto label_A_waiting;
    fi
  }
  :: atomic { chan_k[((i == i) -> 0 : 1)] ! k(i) -> if
    :: sync ? ACK -> {
      d_step {
        i = i + 1;
        reached_home_by = 1;
      }
      goto label_A_home;
    }
    :: sync ? NACK -> goto label_A_waiting;
    fi
  }
  fi
label_A__2:
  if
  :: atomic { chan_success_A[((i == i) -> 0 : 1)] ! success_A(i) -> if
    :: sync ? ACK -> goto label_A_set;
    :: sync ? NACK -> goto label_A__2;
    fi
  }
  fi
label_A_set:
  if
  :: atomic { chan_a_0[0] ? a_0 -> if
    :: true -> {
      d_step {
        i = 1;
        reached_home_by = 2;
      }
      sync ! ACK;
```

```
      goto label_A_home;
    }
    :: else -> sync ! NACK; goto label_A_set;
    fi
  }
  :: atomic {
    d_step {
      i == 0;
      reached_home_by = 3;
    }
    goto label_A_home;
  }
  fi
}

proctype B() {
  int i_0 = 0;
  int k_x_0 = 0;
  int set_B_x = 0;
label_B_home:
  if
  :: atomic { chan_a[((true) -> 0 : 1)] ! a -> if
    :: sync ? ACK -> goto label_B_comm;
    :: sync ? NACK -> goto label_B_home;
    fi
  }
  fi
label_B_comm:
  if
  :: atomic { chan_k[0] ? k(k_x_0) -> if
    :: k_x_0 == i_0 && i_0 < 3 -> {
      d_step {
        i_0 = i_0 + 1;
        k_x_0 = 0;
      }
      sync ! ACK;
      goto label_B_home;
    }
    :: else -> sync ! NACK; k_x_0 = 0; goto label_B_comm;
    fi
  }
  :: atomic { chan_set_B[0] ? set_B(set_B_x) -> if
    :: (set_B_x == 0 && i_0 == 0) || (set_B_x == 1 && i_0 == 0) -> {
      set_B_x = 0;
      sync ! ACK;
      goto label_B_comm;
    }
    :: else -> sync ! NACK; set_B_x = 0; goto label_B_comm;
    fi
  }
  :: atomic { chan_a_0[((true) -> 0 : 1)] ! a_0 -> if
    :: sync ? ACK -> goto label_B_home;
    :: sync ? NACK -> goto label_B_comm;
    fi
  }
  fi
}

proctype closure_env() {
  int success_A_new_i_0 = 0;
label_closure_env_home:
  if
  :: atomic { chan_success_A[0] ? success_A(success_A_new_i_0) -> if
    :: true -> {
      success_A_new_i_0 = 0;
      sync ! ACK;
      goto label_closure_env_home;
    }
    :: else -> sync ! NACK; success_A_new_i_0 = 0; goto label_closure_env_home;
```

```
    fi
  }
  :: atomic { chan_set_A[((true) -> 0 : 1)] ! set_A(1) -> if
    :: sync ? ACK -> goto label_closure_env_home;
    :: sync ? NACK -> goto label_closure_env_home;
    fi
  }
  :: atomic { chan_set_A[((true) -> 0 : 1)] ! set_A(2) -> if
    :: sync ? ACK -> goto label_closure_env_home;
    :: sync ? NACK -> goto label_closure_env_home;
    fi
  }
  :: atomic { chan_set_B[((true) -> 0 : 1)] ! set_B(0) -> if
    :: sync ? ACK -> goto label_closure_env_home;
    :: sync ? NACK -> goto label_closure_env_home;
    fi
  }
  :: atomic { chan_set_B[((true) -> 0 : 1)] ! set_B(1) -> if
    :: sync ? ACK -> goto label_closure_env_home;
    :: sync ? NACK -> goto label_closure_env_home;
    fi
  }
  fi
}

init {
  atomic {
    run A();
    run B();
    run closure_env();
  }
}
```

Listing B.3 – Promela translation of the closed model using the naive translation from Section 4.1.

The actual translation is provided below.

```
mtype {a, set_A, k, success_A, a_0, set_B, ACK, NACK};

chan chan_a[2] = [0] of {mtype};
chan chan_set_A[2] = [0] of {mtype, int};
chan chan_k[2] = [0] of {mtype, int};
chan chan_success_A[2] = [0] of {mtype, int};
chan chan_a_0[2] = [0] of {mtype};
chan chan_set_B[2] = [0] of {mtype, int};
chan sync = [0] of {mtype};

int flag = 0;
bit flags_a[1] = 0;
bit flags_k[1] = 0;
bit flags_set_A[2] = 0;
bit flags_success_A[1] = 0;
bit flags_set_B[2] = 0;
bit flags_a_0[1] = 0;

proctype A() {
  int i = 0;
  int reached_home_by = 4;

  int set_A_x = 0;

  printf("A: [i = %d, reached_home_by = %d]\n", i, reached_home_by);
label_A_home:
  if
  :: atomic { chan_a[0] ? a -> if
    :: true -> {
      d_step {
        for (flag in flags_a) { flags_a[flag] = 0; }
```

```
32          for (flag in flags_set_A) { flags_set_A[flag] = 0; }
33          for (flag in flags_a_0) { flags_a_0[flag] = 0; }
34          for (flag in flags_success_A) { flags_success_A[flag] = 0; }
35          flag = 0;
36          printf("A[label_A_home]: chan_a ? a -> label_A_waiting\n");
37          skip;
38        }
39        sync ! ACK;
40        goto label_A_waiting;
41      }
42      :: else -> sync ! NACK; skip; goto label_A_home;
43      fi
44    }
45    fi
46  label_A_waiting:
47    if
48    :: atomic { chan_set_A[0] ? set_A(set_A_x) -> if
49      :: (set_A_x == 1 || set_A_x == 2) && i == 1 -> {
50        d_step {
51          for (flag in flags_a) { flags_a[flag] = 0; }
52          for (flag in flags_set_A) { flags_set_A[flag] = 0; }
53          for (flag in flags_a_0) { flags_a_0[flag] = 0; }
54          for (flag in flags_k) { flags_k[flag] = 0; }
55          flag = 0;
56          printf("A[label_A_waiting]: chan_set_A ? set_A(%d) -> label_A__2\n",
                □ set_A_x);
57          i = set_A_x + 1;
58          printf("A: [i = %d, reached_home_by = %d]\n", i, reached_home_by);
59          set_A_x = 0;
60        }
61        sync ! ACK;
62        goto label_A__2;
63      }
64      :: else -> sync ! NACK; set_A_x = 0; goto label_A_waiting;
65      fi
66    }
67    :: atomic { chan_k[(!flags_k[0] && (i == i) -> 0 : 1)] ! k(i) -> if
68      :: sync ? ACK -> {
69        d_step {
70          for (flag in flags_a) { flags_a[flag] = 0; }
71          for (flag in flags_set_A) { flags_set_A[flag] = 0; }
72          for (flag in flags_a_0) { flags_a_0[flag] = 0; }
73          for (flag in flags_success_A) { flags_success_A[flag] = 0; }
74          flag = 0;
75          printf("A[label_A_waiting]: chan_k ! k(%d) -> label_A_home\n", i);
76          i = i + 1; reached_home_by = 1;
77          printf("A: [i = %d, reached_home_by = %d]\n", i, reached_home_by);
78        }
79        goto label_A_home;
80      }
81      :: sync ? NACK -> flags_k[0] = 1; goto label_A_waiting;
82      fi
83    }
84    fi
85  label_A__2:
86    if
87    :: atomic { chan_success_A[(!flags_success_A[0] && (i == i) -> 0 : 1)] !
          □ success_A(i) -> if
88      :: sync ? ACK -> {
89        d_step {
90          for (flag in flags_a) { flags_a[flag] = 0; }
91          for (flag in flags_set_A) { flags_set_A[flag] = 0; }
92          for (flag in flags_a_0) { flags_a_0[flag] = 0; }
93          for (flag in flags_k) { flags_k[flag] = 0; }
94          flag = 0;
95          printf("A[label_A__2]: chan_success_A ! success_A(%d) -> label_A_set\n"
                □ , i);
96        }
97        goto label_A_set;
```

```
98      }
99      :: sync ? NACK -> flags_success_A[0] = 1; goto label_A__2;
100     fi
101   }
102   fi
103 label_A_set:
104   if
105   ::  atomic { chan_a_0[0] ? a_0 -> if
106     ::  true  -> {
107       d_step {
108         for (flag in flags_a) { flags_a[flag] = 0; }
109         for (flag in flags_set_A) { flags_set_A[flag] = 0; }
110         for (flag in flags_a_0) { flags_a_0[flag] = 0; }
111         for (flag in flags_success_A) { flags_success_A[flag] = 0; }
112         flag = 0;
113         printf ("A[label_A_set]: chan_a_0 ? a_0 -> label_A_home\n");
114         i = 1; reached_home_by = 2;
115         printf ("A: [i = %d, reached_home_by = %d]\n", i, reached_home_by);
116         skip ;
117       }
118       sync ! ACK;
119       goto label_A_home;
120     }
121     ::  else  -> sync ! NACK; skip ;  goto label_A_set;
122     fi
123   }
124   ::  atomic {
125     d_step {
126       i == 0;
127       for (flag in flags_a) { flags_a[flag] = 0; }
128       for (flag in flags_set_A) { flags_set_A[flag] = 0; }
129       for (flag in flags_k) { flags_k[flag] = 0; }
130       for (flag in flags_success_A) { flags_success_A[flag] = 0; }
131       for (flag in flags_a_0) { flags_a_0[flag] = 0; }
132       flag = 0;
133       printf ("A[label_A_set]: unobservable -> label_A_home\n");
134       reached_home_by = 3;
135       printf ("A: [i = %d, reached_home_by = %d]\n", i, reached_home_by);
136     }
137     goto label_A_home;
138   }
139   fi
140 }
141
142 proctype B() {
143   int  i_0 = 0;
144
145   int  k_x_0 = 0;
146   int  set_B_x = 0;
147
148   printf ("B: [i_0 = %d]\n", i_0);
149 label_B_home:
150   if
151   ::  atomic { chan_a[(!flags_a[0] && (  true ) -> 0 : 1)] ! a ->   if
152     :: sync ? ACK -> {
153       d_step {
154         for (flag in flags_k) { flags_k[flag] = 0; }
155         for (flag in flags_set_B) { flags_set_B[flag] = 0; }
156         flag = 0;
157         printf ("B[label_B_home]: chan_a ! a -> label_B_comm\n");
158       }
159       goto label_B_comm;
160     }
161     :: sync ? NACK -> flags_a[0] = 1;  goto label_B_home;
162     fi
163   }
164   fi
165 label_B_comm:
166   if
```

```
  :: atomic { chan_k[0] ? k(k_x_0) -> if
    :: k_x_0 == i_0 && i_0 < 3 -> {
      d_step {
        for (flag in flags_k) { flags_k[flag] = 0; }
        for (flag in flags_set_B) { flags_set_B[flag] = 0; }
        flag = 0;
        printf("B[label_B_comm]: chan_k ? k(%d) -> label_B_home\n", k_x_0);
        i_0 = i_0 + 1;
        printf("B: [i_0 = %d]\n", i_0);
        k_x_0 = 0;
      }
      sync ! ACK;
      goto label_B_home;
    }
    :: else -> sync ! NACK; k_x_0 = 0; goto label_B_comm;
    fi
  }
  :: atomic { chan_set_B[0] ? set_B(set_B_x) -> if
    :: (set_B_x == 0 && i_0 == 0) || (set_B_x == 1 && i_0 == 0) -> {
      d_step {
        for (flag in flags_k) { flags_k[flag] = 0; }
        for (flag in flags_set_B) { flags_set_B[flag] = 0; }
        for (flag in flags_a) { flags_a[flag] = 0; }
        for (flag in flags_a_0) { flags_a_0[flag] = 0; }
        flag = 0;
        printf("B[label_B_comm]: chan_set_B ? set_B(%d) -> label_B_comm\n",
            □ set_B_x);
        set_B_x = 0;
      }
      sync ! ACK;
      goto label_B_comm;
    }
    :: else -> sync ! NACK; set_B_x = 0; goto label_B_comm;
    fi
  }
  :: atomic { chan_a_0[(!flags_a_0[0] && (true) -> 0 : 1)] ! a_0 -> if
    :: sync ? ACK -> {
      d_step {
        for (flag in flags_k) { flags_k[flag] = 0; }
        for (flag in flags_set_B) { flags_set_B[flag] = 0; }
        flag = 0;
        printf("B[label_B_comm]: chan_a_0 ! a_0 -> label_B_home\n");
      }
      goto label_B_home;
    }
    :: sync ? NACK -> flags_a_0[0] = 1; goto label_B_comm;
    fi
  }
  fi
}

proctype closure_env() {
  int success_A_new_i_0 = 0;

  skip;
label_closure_env_home:
  if
  :: atomic { chan_success_A[0] ? success_A(success_A_new_i_0) -> if
    :: true -> {
      d_step {
        for (flag in flags_success_A) { flags_success_A[flag] = 0; }
        for (flag in flags_set_B) { flags_set_B[flag] = 0; }
        flag = 0;
        skip;
        success_A_new_i_0 = 0;
      }
      sync ! ACK;
      goto label_closure_env_home;
    }
```

```
235       :: else -> sync ! NACK; success_A_new_i_0 = 0; goto label_closure_env_home;
236       fi
237     }
238     :: atomic { chan_set_A[(!flags_set_A[0] && (true) -> 0 : 1)] ! set_A(1) -> if
239       :: sync ? ACK -> {
240         d_step {
241           for (flag in flags_success_A) { flags_success_A[flag] = 0; }
242           for (flag in flags_set_B) { flags_set_B[flag] = 0; }
243           flag = 0;
244           skip;
245         }
246         goto label_closure_env_home;
247       }
248       :: sync ? NACK -> flags_set_A[0] = 1; goto label_closure_env_home;
249       fi
250     }
251     :: atomic { chan_set_A[(!flags_set_A[1] && (true) -> 0 : 1)] ! set_A(2) -> if
252       :: sync ? ACK -> {
253         d_step {
254           for (flag in flags_success_A) { flags_success_A[flag] = 0; }
255           for (flag in flags_set_B) { flags_set_B[flag] = 0; }
256           flag = 0;
257           skip;
258         }
259         goto label_closure_env_home;
260       }
261       :: sync ? NACK -> flags_set_A[1] = 1; goto label_closure_env_home;
262       fi
263     }
264     :: atomic { chan_set_B[(!flags_set_B[0] && (true) -> 0 : 1)] ! set_B(0) -> if
265       :: sync ? ACK -> {
266         d_step {
267           for (flag in flags_success_A) { flags_success_A[flag] = 0; }
268           for (flag in flags_set_A) { flags_set_A[flag] = 0; }
269           flag = 0;
270           skip;
271         }
272         goto label_closure_env_home;
273       }
274       :: sync ? NACK -> flags_set_B[0] = 1; goto label_closure_env_home;
275       fi
276     }
277     :: atomic { chan_set_B[(!flags_set_B[1] && (true) -> 0 : 1)] ! set_B(1) -> if
278       :: sync ? ACK -> {
279         d_step {
280           for (flag in flags_success_A) { flags_success_A[flag] = 0; }
281           for (flag in flags_set_A) { flags_set_A[flag] = 0; }
282           flag = 0;
283           skip;
284         }
285         goto label_closure_env_home;
286       }
287       :: sync ? NACK -> flags_set_B[1] = 1; goto label_closure_env_home;
288       fi
289     }
290     fi
291 }
292 init {
293   printf("TRAIL START\n")
294   atomic {
295     run A();
296     run B();
297     run closure_env();
298   }
299 }
```

Listing B.4 – Promela translation of the closed model using the extended translation from Section 4.2.

Appendix C

Experiment Models

In this chapter we show list the AML models we used in Chapter 5 in full.

C.1 Unit Tests

In Section 5.1 we performed three experiments akin to unit tests. With these tests we provided confidence in the correctness of our mapping. We created three templates that produce AML models. These models all follow the same structure. We call models produced by the first template "simple receivers". Models produced by the second and third template are called "symbolic receivers" and "symbolic senders", respectively. All three templates take parameters □, □, and □ as their parameters. Parameter □ determines the number of processes, □ determines the upper bound for process variable `i` of each process, and □ determines the number of locations in each process. Below we provide an example of each kind of model for □ = 1, □ = 3, and □ = 3.

We start with the simple receiver.

```
external 'ext'
process('proc1') {
  channel('ext') {
    stimulus 'inc'
  }
  var 'i', :integer, 1

  state 'start'
  choice {
    o { goto 'p1' }
    o { goto 'p2' }
    o { goto 'p3' }
  }
  state 'p1'
  receive 'inc', constraint: 'i < 3', update: 'i = i + 1'
  goto 'p1'

  state 'p2'
  receive 'inc', constraint: 'i < 3', update: 'i = i + 1'
  goto 'p2'

  state 'p3'
  receive 'inc', constraint: 'i < 3', update: 'i = i + 1'
  goto 'p3'
```

```
}
```

Listing C.1 – Simple receiver for □ = 1, □ = 3, □ = 3.

The `inc` label is parameterized by integer `x` in the symbolic sender and receiver models. A process of a symbolic receiver models only accept an `inc` label when `x` is equal to the value of its sole process variable, `i`.

Below we list the symbolic receiver using the aforementioned parameters.

```
external 'ext'
process('proc1') {
  channel('ext') {
    stimulus 'inc', { 'x' => :integer }
  }
  var 'i', :integer, 1

  state 'start'
  choice {
    o { goto 'p1' }
    o { goto 'p2' }
    o { goto 'p3' }
  }
  state 'p1'
  choice {
    o { receive 'inc', constraint: 'i == 1 && x == 1', update: 'i = i + 1' }
    o { receive 'inc', constraint: 'i == 2 && x == 2', update: 'i = i + 1' }
  }
  goto 'p1'

  state 'p2'
  choice {
    o { receive 'inc', constraint: 'i == 1 && x == 1', update: 'i = i + 1' }
    o { receive 'inc', constraint: 'i == 2 && x == 2', update: 'i = i + 1' }
  }
  goto 'p2'

  state 'p3'
  choice {
    o { receive 'inc', constraint: 'i == 1 && x == 1', update: 'i = i + 1' }
    o { receive 'inc', constraint: 'i == 2 && x == 2', update: 'i = i + 1' }
  }
  goto 'p3'
}
```

Listing C.2 – Symbolic receiver for □ = 1, □ = 3, □ = 3.

Note that at location there are □ − 1 transitions, of which at most one is enabled. The transitions at each location can be merged into one, like so:

```
state 'p1'
receive 'inc', constraint: '(x == 1 && i == x) || (x == 2 && i == x)', update:
    □ 'i = i + 1'
goto 'p1'
```

However, to ease the creation of our template, we decided to separate these transitions. This does not affect the state space of the resulting Promela program.

Below we list the symbolic sender, again using the aforementioned parameters.

```
external 'ext'
timeout 1

process('proc1') {
  channel('ext') {
```

```
    response 'inc', { 'x' => :integer }
  }
  var 'i', :integer, 1

  state 'start'
  choice {
    o { goto 'p1' }
    o { goto 'p2' }
    o { goto 'p3' }
  }
  state 'p1'
  send 'inc', constraint: 'i < 3 && x == i', update: 'i = i + 1'
  goto 'p1'

  state 'p2'
  send 'inc', constraint: 'i < 3 && x == i', update: 'i = i + 1'
  goto 'p2'

  state 'p3'
  send 'inc', constraint: 'i < 3 && x == i', update: 'i = i + 1'
  goto 'p3'
}
```

Listing C.3 – Symbolic sender for □ = 1, □ = 3, □ = 3.

C.2 The Bounded Retransmission Protocol

For our experiments with the BRP we have created an AML model, and a template. We used the model to verify that the Promela code produced by the prototype for this model satisfies the same properties as the Promela programs provided in [45]. As stated in Section 5.2, we replaced the environment process generated by the prototype with the environment process present from one of the programs in [45]. Consequently, this AML model lacks any logic regarding the generation of random byte sequences. We have listed this model below:

```
external 'ext'
internal 'F', 'G', 'A', 'B', 'ctimeout', 'syncwait'

timeout 1

#############
## SENDER ##
#############
process('sender') {
  channel('ext') {
    stimulus 'req'
    response 'ok'
    response 'idk'
    response 'nok'
  }
  channel('F') {
    response 'f', { 'b_first' => :integer, 'b_last' => :integer, 'ab' => :
        □ integer, 'datum' => :integer }
  }
  channel('B') {
    stimulus 'b_ack'
  }
  channel('ctimeout') {
    stimulus 'ct_shake_l'
    stimulus 'ct_shake_k'
  }
  channel('syncwait') {
    stimulus 'sw_shake_rs'
    response 'sw_shake_sr'
  }
```

```
30
31  var 'cab', :integer, 0   # current alternating bit
32  var 'rc', :integer, 0    # retry counter
33  var 'max', :integer, 3   # max retransmissions (constant)
34  var 'i', :integer, 0     # current chunk
35  var 'n', :integer, 0     # amount of chunks
36  var 'cdatum', :integer, 0 # current chunk data
37
38  # data chunks
39  var 'd1', :integer, 0
40  var 'd2', :integer, 0
41  var 'd3', :integer, 0
42
43  state 'idle'
44  receive 'req'
45  receive 'req', update: 'i = 1; cdatum = d1'
46  goto 'next_frame'
47
48  state 'next_frame'
49  choice {
50    o { send 'f', constraint: 'b_first == 1 && i == 1 && b_last == 1 && i == n
          □ && ab == cab && datum == cdatum', update: 'rc = 0' }
51    o { send 'f', constraint: 'b_first == 0 && i != 1 && b_last == 1 && i == n
          □ && ab == cab && datum == cdatum', update: 'rc = 0' }
52    o { send 'f', constraint: 'b_first == 1 && i == 1 && b_last == 0 && i != n
          □ && ab == cab && datum == cdatum', update: 'rc = 0' }
53    o { send 'f', constraint: 'b_first == 0 && i != 1 && b_last == 0 && i != n
          □ && ab == cab && datum == cdatum', update: 'rc = 0' }
54  }
55  goto 'wait_ack'
56
57  state 'wait_ack'
58  choice {
59    o {
60      receive 'b_ack', update: 'cab = (cab + 1) % 2'
61      goto 'success'
62    }
63    o {
64      choice {
65        o { receive 'ct_shake_l' }
66        o { receive 'ct_shake_k' }
67      }
68      choice {
69        o {
70          tau constraint: 'rc < max', update: 'rc = rc + 1'
71          choice {
72            o { send 'f', constraint: 'b_first == 1 && i == 1 && b_last == 1 &&
                    □  i == n && ab == cab && datum == cdatum' }
73            o { send 'f', constraint: 'b_first == 0 && i != 1 && b_last == 1 &&
                    □  i == n && ab == cab && datum == cdatum' }
74            o { send 'f', constraint: 'b_first == 1 && i == 1 && b_last == 0 &&
                    □  i != n && ab == cab && datum == cdatum' }
75            o { send 'f', constraint: 'b_first == 0 && i != 1 && b_last == 0 &&
                    □  i != n && ab == cab && datum == cdatum' }
76          }
77          goto 'wait_ack'
78        }
79        o {
80          tau constraint: 'rc >= max'
81          goto 'error'
82        }
83      }
84    }
85  }
86
87  state 'success'
88  choice {
89    o {
90      send 'ok', constraint: 'i == n'
```

```
91        goto 'idle'
92      }
93      o {
94        tau constraint : 'i < n', update: 'i = i + 1'
95        choice {
96          o { tau constraint : 'i == 1', update: 'cdatum = d1' }
97          o { tau constraint : 'i == 2', update: 'cdatum = d2' }
98          o { tau constraint : 'i == 3', update: 'cdatum = d3' }
99        }
100       goto 'next_frame'
101     }
102   }
103
104   state 'error'
105   choice {
106     o {
107       send 'idk', constraint : 'i == n'
108     }
109     o {
110       send 'nok', constraint : 'i != n'
111     }
112   }
113   send 'sw_shake_sr'
114   receive 'sw_shake_rs', update: 'cab = 0'
115   goto 'idle'
116 }
117
118 ###############
119 ## RECEIVER ##
120 ###############
121 process('receiver') {
122   channel('ext') {
123     response 'fst', { 'datum' => :integer }
124     response 'ok', { 'datum' => :integer }
125     response 'nok', { 'datum' => :integer }
126     response 'inc', { 'datum' => :integer }
127   }
128   channel('G') {
129     stimulus 'g', { 'b_first' => :integer, 'b_last' => :integer, 'ab' => :
          integer, 'datum' => :integer }
130   }
131   channel('A') {
132     response 'a_ack'
133   }
134   channel('syncwait') {
135     stimulus 'sw_shake_sr'
136     response 'sw_shake_rs'
137   }
138   var 'rab', : integer, 0        # received alternating bit
139   var 'exp_ab', : integer, 0     # expected alternating bit
140   var 'r_b_first', : integer, 0 # received first bit?
141   var 'r_b_last', : integer, 0  # received last bit?
142   var 'cdatum', : integer, 0     # current datum
143
144   state 'new_file'
145   choice {
146     o {
147       receive 'g', update: 'rab = ab; r_b_first = b_first; r_b_last = b_last;
            cdatum = datum'
148       goto 'first_safe_frame'
149     }
150     o {
151       receive 'sw_shake_sr'
152       send 'sw_shake_rs'
153       goto 'new_file'
154     }
155   }
156
157   state 'first_safe_frame'
```

```
158   tau update: 'exp_ab = rab'
159   goto 'frame_received'
160
161   state 'frame_received'
162   choice {
163     o {
164       tau constraint: 'rab != exp_ab'
165       send 'a_ack'
166       goto 'idle'
167     }
168     o {
169       tau constraint: 'rab == exp_ab'
170       choice {
171         o { send 'fst', constraint: 'r_b_first == 1 && r_b_last == 0 && datum
                □ == cdatum' }
172         o { send 'inc', constraint: 'r_b_first == 0 && r_b_last == 0 && datum
                □ == cdatum' }
173         o { send 'ok', constraint: 'r_b_last == 1 && datum == cdatum' }
174       }
175       goto 'frame_reported'
176     }
177   }
178
179   state 'frame_reported'
180   send 'a_ack', update: 'exp_ab = (exp_ab + 1) % 2'
181   goto 'idle'
182
183   state 'idle'
184   choice {
185     o {
186       receive 'g', update: 'r_b_first = b_first; r_b_last = b_last; rab = ab;
              □ cdatum = datum'
187       goto 'frame_received'
188     }
189     o {
190       receive 'sw_shake_sr'
191       choice {
192         o { tau constraint: 'r_b_last == 1' }
193         o { send 'nok', constraint: 'r_b_last == 0 && datum == 0' }
194       }
195       send 'sw_shake_rs'
196       goto 'new_file'
197     }
198   }
199 }
200
201 ################
202 ## CHANNEL K ##
203 ################
204 process('k') {
205   channel('F') {
206     stimulus 'f', { 'b_first' => :integer, 'b_last' => :integer, 'ab' => :
            □ integer, 'datum' => :integer }
207   }
208   channel('G') {
209     response 'g', { 'b_first' => :integer, 'b_last' => :integer, 'ab' => :
            □ integer, 'datum' => :integer }
210   }
211   channel('ctimeout') {
212     response 'ct_shake_k'
213   }
214
215   var 'r_b_first', :integer, 0 # received first bit?
216   var 'r_b_last', :integer, 0  # received last bit?
217   var 'r_ab', :integer, 0      # received alternating bit
218   var 'r_datum', :integer, 0   # received datum
219
220   state 'home'
221   receive 'f', update: 'r_b_first = b_first; r_b_last = b_last; r_ab = ab;
```

```
      □ r_datum = datum'
  choice {
    o {
      send 'g', constraint: 'b_first == r_b_first && b_last == r_b_last && ab
          □ == r_ab && datum == r_datum'
    }
    o {
      tau
      send 'ct_shake_k'
    }
  }
  goto 'home'
}

################
## CHANNEL L ##
################
process('l') {
  channel('A') {
    stimulus 'a_ack'
  }
  channel('B') {
    response 'b_ack'
  }
  channel('ctimeout') {
    response 'ct_shake_l'
  }

  state 'home'
  receive 'a_ack'
  choice {
    o {
      send 'b_ack'
    }
    o { tau
        send 'ct_shake_l'
    }
  }
  goto 'home'
}
```

Listing C.4 – Implementation of the BRP, lacking the generation of random sequences.

AML models created from the template are very similar. Since the environment processes our prototype generates are not capable of generating random sequences, we have moved this responsibility to the sender process. Below, we show the code produced from the template with random sequence maximum length □ = 3, and maximum amount of retransmissions □ = 3.

```
external 'ext'
internal 'F', 'G', 'A', 'B', 'ctimeout', 'syncwait'

timeout 1

#############
## SENDER ##
#############
process('sender') {
  channel('ext') {
    stimulus 'req'
    response 'ok'
    response 'idk'
    response 'nok'
  }
  channel('F') {
    response 'f', { 'b_first' => :integer, 'b_last' => :integer, 'ab' => :
        □ integer, 'datum' => :integer }
```

```
  }
  channel('B') {
    stimulus 'b_ack'
  }
  channel('ctimeout') {
    stimulus 'ct_shake_l'
    stimulus 'ct_shake_k'
  }
  channel('syncwait') {
    stimulus 'sw_shake_rs'
    response 'sw_shake_sr'
  }

  var 'cab', : integer , 0   # current alternating bit
  var 'rc', : integer , 0    # retry counter
  var 'max', : integer , 3   # max retransmissions (constant)
  var 'i', : integer , 0     # current chunk
  var 'n', : integer , 0     # amount of chunks
  var 'cdatum', : integer , 0 # current chunk data

  # data chunks
  var 'd1', : integer , 0
  var 'd2', : integer , 0
  var 'd3', : integer , 0

  state 'idle'
  receive 'req', update: 'd1 = 0; d2 = 0; d3 = 0
  choice {
    o { tau update: 'n = 1'}
    o { tau update: 'n = 2'}
    o { tau update: 'n = 3'}
  }
  choice {
    o { tau constraint : 'n == 1'
        choice {
          o { tau update: 'd1 = 1' }
        }
    }
    o { tau constraint : 'n == 2'
        choice {
          o { tau update: 'd1 = 1' }
          o { tau update: 'd2 = 1' }
        }
    }
    o { tau constraint : 'n == 3'
        choice {
          o { tau update: 'd1 = 1' }
          o { tau update: 'd2 = 1' }
          o { tau update: 'd3 = 1' }
        }
    }
  }
  receive 'req', update: 'i = 1; cdatum = d1'
  goto 'next_frame'

  state 'next_frame'
  choice {
    o { send 'f', constraint : 'b_first == 1 && i == 1 && b_last == 1 && i == n
        ± && ab == cab && datum == cdatum update: 'rc = 0' }
    o { send 'f', constraint : 'b_first == 0 && i != 1 && b_last == 1 && i == n
        ± && ab == cab && datum == cdatum update: 'rc = 0' }
    o { send 'f', constraint : 'b_first == 1 && i == 1 && b_last == 0 && i != n
        ± && ab == cab && datum == cdatum update: 'rc = 0' }
    o { send 'f', constraint : 'b_first == 0 && i != 1 && b_last == 0 && i != n
        ± && ab == cab && datum == cdatum update: 'rc = 0' }
  }
  goto 'wait_ack'

  state 'wait_ack'
```

```
  choice {
    o {
      receive 'b_ack', update: 'cab = (cab + 1) % 2'
      goto 'success'
    }
    o {
      choice {
        o { receive 'ct_shake_l' }
        o { receive 'ct_shake_k' }
      }
      choice {
        o {
          tau constraint: 'rc < max', update: 'rc = rc + 1'
          choice {
            o { send 'f', constraint: 'b_first == 1 && i == 1 && b_last == 1 &&
                □  i == n && ab == cab && datum == cdatum' }
            o { send 'f', constraint: 'b_first == 0 && i != 1 && b_last == 1 &&
                □  i == n && ab == cab && datum == cdatum' }
            o { send 'f', constraint: 'b_first == 1 && i == 1 && b_last == 0 &&
                □  i != n && ab == cab && datum == cdatum' }
            o { send 'f', constraint: 'b_first == 0 && i != 1 && b_last == 0 &&
                □  i != n && ab == cab && datum == cdatum' }
          }
          goto 'wait_ack'
        }
        o {
          tau constraint: 'rc >= max'
          goto 'error'
        }
      }
    }
  }

  state 'success'
  choice {
    o {
      send 'ok', constraint: 'i == n'
      goto 'idle'
    }
    o {
      tau constraint: 'i < n', update: 'i = i + 1'
      choice {
        o { tau constraint: 'i == 1', update: 'cdatum = d1' }
        o { tau constraint: 'i == 2', update: 'cdatum = d2' }
        o { tau constraint: 'i == 3', update: 'cdatum = d3' }
      }
      goto 'next_frame'
    }
  }

  state 'error'
  choice {
    o {
      send 'idk', constraint: 'i == n'
    }
    o {
      send 'nok', constraint: 'i != n'
    }
  }
  send 'sw_shake_sr'
  receive 'sw_shake_rs', update: 'cab = 0'
  goto 'idle'
}

###############
## RECEIVER ##
###############
process('receiver') {
  channel('ext') {
```

```
148     response 'fst', { 'datum' => :integer }
149     response 'ok', { 'datum' => :integer }
150     response 'nok', { 'datum' => :integer }
151     response 'inc', { 'datum' => :integer }
152   }
153   channel('G') {
154     stimulus 'g', { 'b_first' => :integer, 'b_last' => :integer, 'ab' => :
            integer, 'datum' => :integer }
155   }
156   channel('A') {
157     response 'a_ack'
158   }
159   channel('syncwait') {
160     stimulus 'sw_shake_sr'
161     response 'sw_shake_rs'
162   }
163   var 'rab', :integer, 0       # received alternating bit
164   var 'exp_ab', :integer, 0    # expected alternating bit
165   var 'r_b_first', :integer, 0 # received first bit?
166   var 'r_b_last', :integer, 0  # received last bit?
167   var 'cdatum', :integer, 0    # current datum
168
169   state 'new_file'
170   choice {
171     o {
172       receive 'g', update: 'rab = ab; r_b_first = b_first; r_b_last = b_last;
              cdatum = datum'
173       goto 'first_safe_frame'
174     }
175     o {
176       receive 'sw_shake_sr'
177       send 'sw_shake_rs'
178       goto 'new_file'
179     }
180   }
181
182   state 'first_safe_frame'
183   tau update: 'exp_ab = rab'
184   goto 'frame_received'
185
186   state 'frame_received'
187   choice {
188     o {
189       tau constraint: 'rab != exp_ab'
190       send 'a_ack'
191       goto 'idle'
192     }
193     o {
194       tau constraint: 'rab == exp_ab'
195       choice {
196         o { send 'fst', constraint: 'r_b_first == 1 && r_b_last == 0 && datum
                == cdatum' }
197         o { send 'inc', constraint: 'r_b_first == 0 && r_b_last == 0 && datum
                == cdatum' }
198         o { send 'ok', constraint: 'r_b_last == 1 && datum == cdatum' }
199       }
200       goto 'frame_reported'
201     }
202   }
203
204   state 'frame_reported'
205   send 'a_ack', update: 'exp_ab = (exp_ab + 1) % 2'
206   goto 'idle'
207
208   state 'idle'
209   choice {
210     o {
211       receive 'g', update: 'r_b_first = b_first; r_b_last = b_last; rab = ab;
              cdatum = datum'
```

```
      goto 'frame_received'
    }
    o {
      receive 'sw_shake_sr'
      choice {
        o { tau constraint: 'r_b_last == 1' }
        o { send 'nok', constraint: 'r_b_last == 0 && datum == 0' }
      }
      send 'sw_shake_rs'
      goto 'new_file'
    }
  }
}

################
## CHANNEL K ##
################
process('k') {
  channel('F') {
    stimulus 'f', { 'b_first' => :integer, 'b_last' => :integer, 'ab' => :
        integer, 'datum' => :integer }
  }
  channel('G') {
    response 'g', { 'b_first' => :integer, 'b_last' => :integer, 'ab' => :
        integer, 'datum' => :integer }
  }
  channel('ctimeout') {
    response 'ct_shake_k'
  }

  var 'r_b_first', :integer, 0 # received first bit?
  var 'r_b_last', :integer, 0  # received last bit?
  var 'r_ab', :integer, 0      # received alternating bit
  var 'r_datum', :integer, 0   # received datum

  state 'home'
  receive 'f', update: 'r_b_first = b_first; r_b_last = b_last; r_ab = ab;
      r_datum = datum'
  choice {
    o {
      send 'g', constraint: 'b_first == r_b_first && b_last == r_b_last && ab
          == r_ab && datum == r_datum'
    }
    o {
      tau
      send 'ct_shake_k'
    }
  }
  goto 'home'
}

################
## CHANNEL L ##
################
process('l') {
  channel('A') {
    stimulus 'a_ack'
  }
  channel('B') {
    response 'b_ack'
  }
  channel('ctimeout') {
    response 'ct_shake_l'
  }

  state 'home'
  receive 'a_ack'
  choice {
    o {
```

```
277         send 'b_ack'
278       }
279       o { tau
280           send 'ct_shake_1'
281       }
282     }
283     goto 'home'
284   }
```

Listing C.5 – Implemenation of BRP. The sender process is responsible for generating the random sequences.

C.3 Dining Philosophers

For our experiment with the dining philosophers problem we created two templates. One template produces AML models, and the other produces Promela programs. Both templates take □, the number of philosophers (and forks), as their only parameter. The AML models and Promela programs these templates produce follow the same structure: there is a table process managing access to the forks. Each philosopher is represented by a separate process. In order to grab a fork, a philosopher must send a request to the table process. This request is granted only when the fork is free. Similarly, philosophers synchronize with the table process to put forks down. The Promela template produces Promela programs one would make following this structure. For this reason we call these programs "reference programs". We call the Promela programs produced by our pipeline for the AML models generated by the AML template "prototype programs".

Below we list both kinds of programs for □ = 3. We start with the reference program.

```
1  #define send_grab(index) comm[0] ! grab(index);
2  #define send_release(index) comm[0] ! release(index);
3  #define receive_grab(index) atomic { comm[(!forks[index] -> 0 : 1)] ? grab(
       □ index) -> forks[index] = true; goto home; }
4  #define receive_release(index) atomic { comm[(forks[index] -> 0 : 1)] ? release
       □ (index) -> forks[index] = false; goto home; }
5
6  mtype { grab, release }
7
8  // 0 = success; 1 = failure receive
9  chan comm [2] = [0] of {mtype, int};
10
11 active proctype table() {
12   bool forks[3] = false;
13 home:
14   if
15   :: receive_grab(0)
16   :: receive_grab(1)
17   :: receive_grab(2)
18   :: receive_release(0)
19   :: receive_release(1)
20   :: receive_release(2)
21   fi
22 }
23
24 proctype phil(int first; int second) {
25 thinking:
26   send_grab(first);
27   send_grab(second);
28   send_release(first);
29   send_release(second);
30   goto thinking;
```

```
}

init {
  atomic { run phil(1, 0);  run phil(1, 2);  run phil(2, 0); }
}
```

Listing C.6 – Reference Promela program for the dining philosophers problem with □ = 3.

Below we list the prototype program.

```
internal 'int'

process('table') {
  channel('int') {
    stimulus 'pick_up0', { 'index' => :integer }
    stimulus 'put_down0', { 'index' => :integer }
    stimulus 'pick_up1', { 'index' => :integer }
    stimulus 'put_down1', { 'index' => :integer }
    stimulus 'pick_up2', { 'index' => :integer }
    stimulus 'put_down2', { 'index' => :integer }
  }

  var 'fork0', :boolean, false
  var 'fork1', :boolean, false
  var 'fork2', :boolean, false

  state 'home'
  choice {
    o {
      receive 'pick_up0', constraint: 'index == 0 && !fork0', update: 'fork0 =
          □ true'
      goto 'home'
    }
    o {
      receive 'pick_up0', constraint: 'index == 1 && !fork1', update: 'fork1 =
          □ true'
      goto 'home'
    }
    o {
      receive 'pick_up1', constraint: 'index == 1 && !fork1', update: 'fork1 =
          □ true'
      goto 'home'
    }
    o {
      receive 'pick_up1', constraint: 'index == 2 && !fork2', update: 'fork2 =
          □ true'
      goto 'home'
    }
    o {
      receive 'pick_up2', constraint: 'index == 2 && !fork2', update: 'fork2 =
          □ true'
      goto 'home'
    }
    o {
      receive 'pick_up2', constraint: 'index == 0 && !fork0', update: 'fork0 =
          □ true'
      goto 'home'
    }

    o {
      receive 'put_down0', constraint: 'index == 0 && fork0', update: 'fork0 =
          □ false'
      goto 'home'
    }
    o {
      receive 'put_down0', constraint: 'index == 1 && fork1', update: 'fork1 =
          □ false'
      goto 'home'
```

```
51      }
52      o {
53        receive 'put_down1', constraint: 'index == 1 && fork1', update: 'fork1 =
              □ false'
54        goto 'home'
55      }
56      o {
57        receive 'put_down1', constraint: 'index == 2 && fork2', update: 'fork2 =
              □ false'
58        goto 'home'
59      }
60      o {
61        receive 'put_down2', constraint: 'index == 2 && fork2', update: 'fork2 =
              □ false'
62        goto 'home'
63      }
64      o {
65        receive 'put_down2', constraint: 'index == 0 && fork0', update: 'fork0 =
              □ false'
66        goto 'home'
67      }
68    }
69  }
70
71  process('phil0') {
72    channel('int') {
73      response 'pick_up0', { 'index' => :integer }
74      response 'put_down0', { 'index' => :integer }
75    }
76
77    state 'think'
78    send 'pick_up0', constraint: 'index == 1'
79    send 'pick_up0', constraint: 'index == 0'
80    send 'put_down0', constraint: 'index == 1'
81    send 'put_down0', constraint: 'index == 0'
82    goto 'think'
83  }
84
85  process('phil1') {
86    channel('int') {
87      response 'pick_up1', { 'index' => :integer }
88      response 'put_down1', { 'index' => :integer }
89    }
90
91    state 'think'
92    send 'pick_up1', constraint: 'index == 1'
93    send 'pick_up1', constraint: 'index == 2'
94    send 'put_down1', constraint: 'index == 1'
95    send 'put_down1', constraint: 'index == 2'
96    goto 'think'
97  }
98
99  process('phil2') {
100   channel('int') {
101     response 'pick_up2', { 'index' => :integer }
102     response 'put_down2', { 'index' => :integer }
103   }
104
105   state 'think'
106   send 'pick_up2', constraint: 'index == 2'
107   send 'pick_up2', constraint: 'index == 0'
108   send 'put_down2', constraint: 'index == 2'
109   send 'put_down2', constraint: 'index == 0'
110   goto 'think'
111 }
```

Listing C.7 – Our implementation of the dining philosophers problem in AML with □ = 3 philosophers.

C.4 Tower of Hanoi

Our experiment with the tower of Hanoi problem followed the same setup as our experiment with the dining philosophers problem. We have created two templates. One template produces AML models, and the other produces Promela programs. Both templates take □, the number of rings, as their only parameter. We call the Promela programs produced by our pipeline for the AML models "prototype programs", and we call the Promela programs produced by the other template "reference programs". Both program kinds follow the same structure: one process registers the position of each ring. In order to move a ring from one pole to another, an environment process sends a move request. If the move is valid, then the processes synchronize. The reference programs represent Promela programs one would write manually following the same structure.

Below we list both kinds of program for □ = 3. We start with the reference program.

```
#define receive(a, b) atomic { comm[(a[height_ ## a - 1] < b[height_ ## b - 1]
    □ -> 0 : 1)] ? a ## b -> b[height_ ## b] = a[height_ ## a - 1]; height_ ##
    □  b = height_ ## b + 1; a[height_ ## a - 1] = 0; height_ ## a = height_
    □ ## a - 1 }
#define send(msg) atomic { comm[0] ! msg; }

mtype { ab, ba, ac, ca, bc, cb };

// 0 = success; 1 = failure
chan comm[2] = [0] of {mtype}

active proctype hanoi() {
  int height_a = 4;
  int a[4] = 0;
  a[0] = 4;
  a[1] = 3;
  a[2] = 2;
  a[3] = 1;
  int height_b = 1;
  int b[4] = 0;
  b[0] = 4;
  int height_c = 1;
  int c[4] = 0;
  c[0] = 4;

  do
  :: receive(a, b)
  :: receive(a, c)
  :: receive(b, a)
  :: receive(b, c)
  :: receive(c, a)
  :: receive(c, b)
  od;
}

active proctype env() {
  do
  :: send(ab)
  :: send(ac)
  :: send(ba)
  :: send(bc)
  :: send(ca)
  :: send(cb)
  od;
}
```

Listing C.8 – Reference Promela program for the tower of Hanoi problem with □ = 3.

Below we list the prototype program.

```
external  'ext'

process('hanoi') {
  channel('ext') {
    stimulus  'ab'
    stimulus  'ba'
    stimulus  'ac'
    stimulus  'ca'
    stimulus  'bc'
    stimulus  'cb'
  }

  var 'height_a', : integer , 3
  var 'a1', : integer , 3
  var 'a2', : integer , 2
  var 'a3', : integer , 1
  var 'height_b', : integer , 0
  var 'b1', : integer , 0
  var 'b2', : integer , 0
  var 'b3', : integer , 0
  var 'height_c', : integer , 0
  var 'c1', : integer , 0
  var 'c2', : integer , 0
  var 'c3', : integer , 0

  state  'home'
  choice {
    o {
      receive  'ab', constraint : 'height_a == 1 && height_b == 0', update: '
          ± height_a = 0; b1 = a1; a1 = 0; height_b = 1'
      goto 'home'
    }
    o {
      receive  'ab', constraint : 'height_a == 1 && height_b == 1 && a1 < b1'
          ± update: 'height_a = 0; b2 = a1; a1 = 0; height_b = 2'
      goto 'home'
    }
    o {
      receive  'ab', constraint : 'height_a == 1 && height_b == 2 && a1 < b2'
          ± update: 'height_a = 0; b3 = a1; a1 = 0; height_b = 3'
      goto 'home'
    }
    o {
      receive  'ab', constraint : 'height_a == 2 && height_b == 0', update: '
          ± height_a = 1; b1 = a2; a2 = 0; height_b = 1'
      goto 'home'
    }
    o {
      receive  'ab', constraint : 'height_a == 2 && height_b == 1 && a2 < b1'
          ± update: 'height_a = 1; b2 = a2; a2 = 0; height_b = 2'
      goto 'home'
    }
    o {
      receive  'ab', constraint : 'height_a == 3 && height_b == 0', update: '
          ± height_a = 2; b1 = a3; a3 = 0; height_b = 1'
      goto 'home'
    }
    o {
      receive  'ac', constraint : 'height_a == 1 && height_c == 0', update: '
          ± height_a = 0; c1 = a1; a1 = 0; height_c = 1'
      goto 'home'
    }
    o {
      receive  'ac', constraint : 'height_a == 1 && height_c == 1 && a1 < c1'
          ± update: 'height_a = 0; c2 = a1; a1 = 0; height_c = 2'
      goto 'home'
    }
    o {
```

```
61        receive 'ac', constraint: 'height_a == 1 && height_c == 2 && a1 < c2',
              update: 'height_a = 0; c3 = a1; a1 = 0; height_c = 3'
62        goto 'home'
63      }
64      o {
65        receive 'ac', constraint: 'height_a == 2 && height_c == 0', update: '
              height_a = 1; c1 = a2; a2 = 0; height_c = 1'
66        goto 'home'
67      }
68      o {
69        receive 'ac', constraint: 'height_a == 2 && height_c == 1 && a2 < c1',
              update: 'height_a = 1; c2 = a2; a2 = 0; height_c = 2'
70        goto 'home'
71      }
72      o {
73        receive 'ac', constraint: 'height_a == 3 && height_c == 0', update: '
              height_a = 2; c1 = a3; a3 = 0; height_c = 1'
74        goto 'home'
75      }
76      o {
77        receive 'ba', constraint: 'height_b == 1 && height_a == 0', update: '
              height_b = 0; a1 = b1; b1 = 0; height_a = 1'
78        goto 'home'
79      }
80      o {
81        receive 'ba', constraint: 'height_b == 1 && height_a == 1 && b1 < a1',
              update: 'height_b = 0; a2 = b1; b1 = 0; height_a = 2'
82        goto 'home'
83      }
84      o {
85        receive 'ba', constraint: 'height_b == 1 && height_a == 2 && b1 < a2',
              update: 'height_b = 0; a3 = b1; b1 = 0; height_a = 3'
86        goto 'home'
87      }
88      o {
89        receive 'ba', constraint: 'height_b == 2 && height_a == 0', update: '
              height_b = 1; a1 = b2; b2 = 0; height_a = 1'
90        goto 'home'
91      }
92      o {
93        receive 'ba', constraint: 'height_b == 2 && height_a == 1 && b2 < a1',
              update: 'height_b = 1; a2 = b2; b2 = 0; height_a = 2'
94        goto 'home'
95      }
96      o {
97        receive 'ba', constraint: 'height_b == 3 && height_a == 0', update: '
              height_b = 2; a1 = b3; b3 = 0; height_a = 1'
98        goto 'home'
99      }
100     o {
101       receive 'bc', constraint: 'height_b == 1 && height_c == 0', update: '
              height_b = 0; c1 = b1; b1 = 0; height_c = 1'
102       goto 'home'
103     }
104     o {
105       receive 'bc', constraint: 'height_b == 1 && height_c == 1 && b1 < c1',
              update: 'height_b = 0; c2 = b1; b1 = 0; height_c = 2'
106       goto 'home'
107     }
108     o {
109       receive 'bc', constraint: 'height_b == 1 && height_c == 2 && b1 < c2',
              update: 'height_b = 0; c3 = b1; b1 = 0; height_c = 3'
110       goto 'home'
111     }
112     o {
113       receive 'bc', constraint: 'height_b == 2 && height_c == 0', update: '
              height_b = 1; c1 = b2; b2 = 0; height_c = 1'
114       goto 'home'
115     }
```

```
116     o {
117       receive 'bc', constraint: 'height_b == 2 && height_c == 1 && b2 < c1',
              update: 'height_b = 1; c2 = b2; b2 = 0; height_c = 2'
118       goto 'home'
119     }
120     o {
121       receive 'bc', constraint: 'height_b == 3 && height_c == 0', update: '
              height_b = 2; c1 = b3; b3 = 0; height_c = 1'
122       goto 'home'
123     }
124     o {
125       receive 'ca', constraint: 'height_c == 1 && height_a == 0', update: '
              height_c = 0; a1 = c1; c1 = 0; height_a = 1'
126       goto 'home'
127     }
128     o {
129       receive 'ca', constraint: 'height_c == 1 && height_a == 1 && c1 < a1',
              update: 'height_c = 0; a2 = c1; c1 = 0; height_a = 2'
130       goto 'home'
131     }
132     o {
133       receive 'ca', constraint: 'height_c == 1 && height_a == 2 && c1 < a2',
              update: 'height_c = 0; a3 = c1; c1 = 0; height_a = 3'
134       goto 'home'
135     }
136     o {
137       receive 'ca', constraint: 'height_c == 2 && height_a == 0', update: '
              height_c = 1; a1 = c2; c2 = 0; height_a = 1'
138       goto 'home'
139     }
140     o {
141       receive 'ca', constraint: 'height_c == 2 && height_a == 1 && c2 < a1',
              update: 'height_c = 1; a2 = c2; c2 = 0; height_a = 2'
142       goto 'home'
143     }
144     o {
145       receive 'ca', constraint: 'height_c == 3 && height_a == 0', update: '
              height_c = 2; a1 = c3; c3 = 0; height_a = 1'
146       goto 'home'
147     }
148     o {
149       receive 'cb', constraint: 'height_c == 1 && height_b == 0', update: '
              height_c = 0; b1 = c1; c1 = 0; height_b = 1'
150       goto 'home'
151     }
152     o {
153       receive 'cb', constraint: 'height_c == 1 && height_b == 1 && c1 < b1',
              update: 'height_c = 0; b2 = c1; c1 = 0; height_b = 2'
154       goto 'home'
155     }
156     o {
157       receive 'cb', constraint: 'height_c == 1 && height_b == 2 && c1 < b2',
              update: 'height_c = 0; b3 = c1; c1 = 0; height_b = 3'
158       goto 'home'
159     }
160     o {
161       receive 'cb', constraint: 'height_c == 2 && height_b == 0', update: '
              height_c = 1; b1 = c2; c2 = 0; height_b = 1'
162       goto 'home'
163     }
164     o {
165       receive 'cb', constraint: 'height_c == 2 && height_b == 1 && c2 < b1',
              update: 'height_c = 1; b2 = c2; c2 = 0; height_b = 2'
166       goto 'home'
167     }
168     o {
169       receive 'cb', constraint: 'height_c == 3 && height_b == 0', update: '
              height_c = 2; b1 = c3; c3 = 0; height_b = 1'
170       goto 'home'
```

```
171         }
172       }
173   }
```

Listing C.9 – AML model for the tower of Hanoi problem with □ = 3.