Mestrado em Engenharia Informática
Dissertação/Estágio
Relatório Final

Geração Automática de Código Fonte a Partir de Modelos Formais

Miguel António Rodrigues Lopes Martins
marm@student.dei.uc.pt

Orientador:

Raul André Brajczewski Barbosa

Data: 4 de Setembro de 2013

FCTUC DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Abstract

The work presented/associated with this document relates to an area of computer science that is described as the generation of source code from the specification of a formal model, which shall be referred to as “Code Generation from Formal Models”. This area is associated with two background areas, one of which being the area of “Model Checking”.

Model Checking, as described by Edmund M. Clarke et al.[1], is “a technique for verifying finite state concurrent systems such as sequential circuit designs and communication protocols”. This technique is appropriate for distributed and concurrent systems, since it aids developers in minimizing certain types of risks, such as the possibility that a deadlock will occur in the system at some point in time, preventing further progress, or the occurrence of a race condition.

Given that *only a model of a system is verified, but not the system itself*, it naturally follows that it would be useful to generate source code from the model specification. This work thus encompasses the generation of source code from PROMELA models, verified by the *Spin* model checker.

In its essence, this project attempts to answer the following question: is it possible to generate runnable source code from PROMELA models related to *round-based consensus protocols*? The short answer to it is “yes, with limitations”.

Keywords

“Java”, “JavaCC”, “Model checking”, “Network simulation”, “OMNeT++”, “PROMELA”, “PROMNeT++”, “Round-based consensus protocols”, “Source code generation”, “Spin”

Contents

Chapter 1

Introduction

It is widely known amongst software engineers today that distributed systems play an integral role on delivering reliable services to real-life customers all around the globe. From ATMs to government databases, these systems provide end users with enhanced reliability, shorter response times, and decreased chances of downtime.

It is imperative that these systems not only function properly under normal circumstances, but also be sufficiently robust to, in the terms of Avizienis et al., “deliver service that can justifiably be trusted”.[2] To achieve this goal, developers must firmly ensure that they operate under correct algorithms. “Correct”, in this context, means that, on one hand, for every possible input and execution, the resulting output must match the expected output. On the other hand, the system must make progress, and not halt indefinitely. Threats such as deadlocks and race conditions must, therefore, be taken into account while coding, as they impede that the system is correct under this definition when present.

Algorithm correctness is, therefore, key to produce a high-quality distributed system. Fortunately, a wide variety of techniques exist for this purpose. These include, but are not limited to, unit testing, model checking, and software inspections. Unit testing, although widely used in the software industry, is a technique that tests portions of the software under a very limited (when compared to the state space of the entire application) set of test cases; it is, therefore, classified as a non-exhaustive testing method. The main flaw with this method is, as pointed out by E. Dijkstra [3], that “non-exhaustive testing can only show the presence of errors, not their absence”. This is due to the fact that, typically, an application’s execution path possesses a rather large branching factor, often resulting in millions or billions of possible combinations.

In contrast, Model Checking is a technique “developed independently by Clarke and Emerson and by Queille and Sifakis in early 1980’s”, whose “verification procedure is an exhaustive search of the state space of the design”. [4] This latter property is key to ensuring software correctness, for it determines that every possible execution path does not yield a violation of the intended protocol(s). Thus, Model Checking belongs to the category of exhaustive testing techniques, and the aforementioned flaw does not apply to it.

Presently, several Model Checking tools exist, one of them being *Spin* [1], an open-source, LTL

[1]`http://spinroot.com/`

model checking tool written entirely in ANSI standard C, and thus highly portable. As demonstrated in the work of Raul Barbosa and Johan Karlsson, a machine with a 3.2 GHz Pentium 4 processor and 1 GB of RAM is able to perform computation at a ratio over 10^4 states per second, and have enough memory to accommodate over 10^8 states, on average, using Spin.[5] These results date back to 2008; nowadays, machines with 4 GB of RAM are becoming more common, but processor speeds are typically inferior to 3.2 GHz, with an average slightly above 2 GHz per core. Luckily, starting from version 5, Spin offers support for multi-core machines [2] which, at this time, are very common. Thus, today, Spin should be able to compute at a ratio of (close to) 10^5 states per second while still maintaining the total number of states at above 10^8. Spin's performance is, thus, quite high.

With Spin, a system is modelled using using PROMELA, a language which is both unambiguous and clear. Unfortunately, while Model Checking is superior to unit testing with regard to exhaustiveness, it has one main flaw of its own: performing Model Checking only guarantees that a particular model complies with its specification; the system itself, and its code, remain unverified, even if developers base the system's code entirely on the model.

To overcome this flaw, this work deals with the automatic generation of runnable source code from a model's specification, and in particular, specifications written in PROMELA. This implies, of course, that we need a software tool that translates PROMELA code to a given target programming language, such as C, C++ or Java. The resulting runnable source code may then be used to further ensure that the system works as desired.

To this end, Miguel Martins (this work's author), has developed a Java-based tool that takes a PROMELA specification (typically a file with a .pml extension) as input and attempts to generate runnable C++ source code based on the specification's PROMELA code, as part of an internship at the University of Coimbra's Department of Informatics Engineering, for the academic year of 2012-2013. Said C++ source code can then be compiled and simulated via OMNeT++ [3] by the user.

This tool is named **PROMNeT++**, and is hosted over at Google Code, along with its source code (main URL: `https://code.google.com/p/promnetpp/`). Both PROMNeT++ itself, in binary form, and its sources, are released under the well-known MIT License [4], which should be permissive enough for anyone to modify and improve the tool as needed, free of charge. Alternatively, if deemed more suitable, developers interested in this project's area of research should be able to construct their own tool from scratch, reusing PROMNeT++'s Java source code as needed (again, free of charge).

More than just generating source code, however, PROMNeT++ was built for one primary purpose, which was to come up with an answer to the following question: *is it possible, for a specific domain of protocols, round-based consesus protocols, to generate runnable code from models pertaining to that domain*? Given that PROMNeT++ is distributed with two round-based protocols and successfully generates runnable C++ code from them, the answer to that question certainly seems to be "yes". However, it should be noted that PROMNeT++ is, by no means, a flawless source code generation tool, and does, in fact, have its own set of limitations,

[2]`http://www.spinroot.com/spin/multicore/V5_Readme.html`
[3]`http://www.omnetpp.org/`
[4]`http://opensource.org/licenses/mit-license.php`

which will be adequately described further in this document.

Ideally, we would like to have produced a tool that would translate any PROMELA specification to C++ code, but restricting the specification domain to round-based consensus protocols was ultimately necessary, given that, first and foremost, a model is a heavy abstraction of a given system, meaning that a large amount of implementation details are either simplified or even omitted.

Models of round-based consensus protocols were chosen (as opposed to other types of models), due to the fact that:

- Round-based consensus protocols have a well-defined structure, with a very specific sequence of procedures; as will be shown further in this document, every node in the protocol makes calls to procedures named *begin_round*, *compute_message*, *send_to_all*, *wait_to_receive*, *state_transition*, and *end_round* within an infinite loop. This structure remains constant across all protocols, and is derived from Eli Gafni's pseudo-code within his article on *Round-by-Round Fault Detectors.*[6]
- The literature on round-based consensus protocols is vast; Google Scholar alone yields 36 results on "round-based consensus" at the time of this document's writing. Round-based consensus protocols should, therefore, be specific enough for carrying out the task of automatic source code generation, while being general enough to be useful to the programming community as a whole.

This document attempts to detail the author's research and methodology efforts. Chapter 2 lists several candidates for software tools relevant to this project, and presents additional work on the generation of source code from formal models. Chapter 3 enunciates this work's objectives, reveals the chosen software tools used to construct PROMNeT++, describes PROMNeT++'s architecture and mechanisms in greater detail, and shows how to collect experimental data. Chapter 4 shows the requirements analysis for PROMNeT++, and shows which of the gathered requirements have been met.

Chapter 5 defines PROMNeT++'s risk analysis through the concept of "threshold of success", and what could have impeded success during the second semester of the internship. Chapter 6 details the results of translating the *OneThirdRule* and *1-of-N* protocols that come packaged with PROMNeT++. Chapter 7 discusses PROMNeT++'s limitations as a software tool, and proposes a few aspects to overcome them in the future. Chapter 8 compares the initial work plan (drafted during the first semester) to the actual, real work the author carried out during the second semester. Finally, chapter 9 captures the key points presented in the previous chapters, and concludes this document.

Chapter 2

State of the art

It is no mere coincidence that PROMNeT++ is written in Java, operates on models written in PROMELA (and thus, runnable in Spin), and generates C++ code that uses, in part, OMNeT++'s API, and is meant to be compiled and run under OMNeT++ itself; prior to PROMNeT++'s development (as is customary with projects developed by internship students at University of Coimbra's Department of Informatics Engineering), a considerable amount of research was carried out.

This project encompasses three types of software tools:

- *Model checking tools.* Given that this project consists on generating runnable source code from formal models, a model checking tool is inherently necessary. Typically, a model is written in a certain modeling language, much like how a computer program is written in a programming language. Of course, this implies that when one chooses a specific model checking tool, they're also inherently choosing a modeling language to work with. For this project, *Spin* is the model checking tool of choice, meaning that PROMELA is the language from which to generate runnable source code.

- *Parsing/language recognition tools.* It should come as no surprise that, to translate a PROMELA specification to any other language, it is necessary to perform parsing on said specification. This is, in fact, what many compilers today do: they take one or more input files written in a certain programming language, and convert the source code into a sequence of tokens (lexical analysis). Then, a parser attempts to match this sequence with a set of grammar rules.

  One approach to building a compiler is to associate each grammar rule with a semantic action, and ultimately build an *Abstract Syntax Tree* out of the various semantic actions. Finally, this Abstract Syntax Tree may be traversed, often multiple times, in order to produce output, typically either assembly code or machine code.

  PROMNeT++ operates under this methodology, building an Abstract Syntax Tree out of PROMELA source code, then traversing it to produce C++ code as output. In this sense, PROMNeT++ can be seen as a *PROMELA to C++ compiler.*

- *Network simulators/network simulation environments.* It should naturally follow that the generated source code, as described above, must be utilized in some way. Round-based consensus protocols, which are within the scope of this project, essentially consist of multiple nodes exchanging messages between each other, in order to reach a decision. It is therefore of interest to see the various nodes interacting/exchanging messages with each other. Thus, a network simulator or simulation environment becomes necessary.

Research efforts were conducted during the first semester of the internship to determine, for each of the above three types, which tool is the most adequate for the project's goals. Below is the result of these efforts.

2.1 Software tools

2.1.1 Model checking

Spin

Spin is regarded as being amongst the most popular model-checking tools [7]. Designed for "efficient verification system for models of distributed software systems"[8], it uses PROMELA (Process Meta Language) as the language for the specification of formal models. In PROMELA, it is possible to specify Linear Temporal Logic (LTL) properties by using its *ltl* statement. As an example, the Spin distribution to date (version 6.2.5) comes with several examples written in PROMELA, one of them corresponding to a model for a leader election protocol. This example contains, among others, the statement

```
{ltl p1 { <>[] (nr_leaders == 1) }}
```

where "nr_leaders" variable represents the number of leaders in the system. The meaning of the above statement is that, eventually (i.e. at some point in time), the condition *nr_leaders == 1* should become true, and when it does, it should never evaluate to false at any further point in time. In turn, this roughly means that, once a leader has been elected, this decision cannot be revoked; furthermore, once a leader has been elected, no other process can be elected for leadership.

Although Spin uses LTL properties for model correctness, assertions also exist in PROMELA. In fact, the aforementioned example contains the following sequence of statements

```
printf("MSC: LEADER\n");
nr_leaders++;
assert(nr_leaders == 1)
```

which are executed when it has been determined that a leader has been chosen. Naturally, this means that when the "nr_leaders" variable is incremented by one unit, it must then immediately be equal to 1, which in turn means that the value 0 must be the initial value for that variable.

SMV

SMV (Symbolic Model Verifier) is "a tool for checking finite state systems against specifications in the temporal logic CTL" [9] from the Carnegie Mellon University. Binaries for several platforms and source code are distributed in the official page [1]. However, at present, the binaries for Windows NT-based operating systems date back to the year of 1998, the latest source code distribution dating back to 2001; thus, both distribution formats could be considered as being quite outdated.

Despite this, extensions of Carnegie Mellon's SMV tool have been produced, in particular *NuSMV*[2] and *Cadence SMV*[3]. The license for CMU's SMV is not known; in contrast, NuSMV is fully open source, and is distributed under the GNU Lesser General Public License 2.1 (LGPL). NuSMV has been successfully used to model and verify the well-known Sliding Window Protocol (which is used, for instance, in TCP for delivering packets in the correct order), which is shown in a paper by Sinha et al. [10] Interestingly, NuSMV has also been used in the domain of Artificial Intelligence: Vishal, Gugwad and Singh have modelled and verified a Multi Agent System related to the management of traffic flow [11].

Cadence SMV, unlike NuSVM, is closed source, and is instead distributed as a Cadence Berkeley Labs Research Software tool, with its own License Agreement; to obtain it, a registration form must be filled accordingly, and one must agree to the License Agreement's terms by pressing the "I Agree" button, located on the bottom of the form[4]. Presently, the latest release is dated October 11, 2002. Despite not being as outdated as CMU's SMV, it is by far more outdated than NuSVM, which dates back to the year of 2011[5].

UPPAAL

Development efforts between Sweden's Uppsala University and Denmark's Aalborg University gave origin to *UPPAAL*, a "toolbox for verification of real-time systems", which "has been applied successfully in case studies ranging from communication protocols to multimedia applications" [12]. The tool itself is written in C++, with a graphical user interface written in Java.

Interestingly, UPPAAL has been used, in conjunction with SPIN, to verify aspects of the Lightweight Underlay Network Ad-hoc Routing protocol (LUNAR), as per the work of Wibling et al. [13]; the authors of this work constructed an UPPAAL model "in order to check timing requirements of LUNAR", one of their objectives being "to achieve an optimal balance that keeps the data packet delivery times as low as possible".

Alloy

Alloy is the name given to not only to a declarative modeling language, but also to the symbolic model checking tool itself. Nimiya et al. have published a paper where they describe the Alloy language as "a simple structural modelling language supported by Alloy analyzer", "used to express complex structural constraints and behaviour" and "based on the first-order logic that allows a user to model a system by abstracting key characteristics of that system"; in the very

[1] `http://www.cs.cmu.edu/~modelcheck/smv.html`

[2] `http://nusmv.fbk.eu/`

[3] `http://www.kenmcmil.com/smv.html`

[4] `http://w2.cadence.com/webforms/cbl_software/index.aspx`

[5] *NuSVM ChangeLog* 2011-10-31 12:00:00 NuSMV team <nusmv@fbk.eu>* === Released version 2.5.4 ===

same paper, they use Alloy to check the consistency of UML diagrams [14].

Possibly one of the most interesting used of the Alloy framework, however, is contained in a paper by Pai et al., where the well-known OAuth 2.0 protocol, used by the popular social network Facebook, is formally verified [15].

Similar tools - KeY

Unlike the tools presented so far, *KeY*[6] is not classified as a model checker, but rather as a deductive verification tool. KeY allows for formal verification of Java programs/code fragments, using either JML or OCL annotations present in the code. For example, a method for computing the factorial of an integer could be given by[7]:

```
public class Fac {
    /* @preconditions n >= 0
    @postconditions result > 0
    */
public static int fac(int n) {
    if (n == 0) return 1;
    else return (n * fac(n - 1));
} }
```

Extensions and/or adaptations of KeY exist, one of them being KeYmaera, “an automated and interactive theorem prover for a natural specification and verification logic for hybrid systems”[8].

2.1.2 Parsing/language recognition

Flex and Bison

Flex and *Bison* were used in the past to implement an ASN.1 (Abstract Syntax Notation One) to C/C++ module compiler, as per the work of *Michael Sample and Gerald Neufeld*[17]. According to said work, this compiler, which goes by the name of *snacc*, successfully implements a vast set of types and features, from basic data types such as BOOLEAN, INTEGER and REAL, to structured types such as SET OF and SEQUENCE OF, going even as far as generating code for memory and error management. Also worth mentioning here, *Flex* and *Bison* can also be used to parse the well-known relational database language, SQL, as demonstrated in John Levine’s book *flex & bison* [18], and thus are powerful enough to translate SQL to another language.

JavaCC and JJTree

JavaCC, also as mentioned above, can also be used for source code generation, and it was, in fact, “initially developed by Sun Microsystems” [19], the company formerly behind the design

[6]`http://www.key-project.org/`

[7]Example taken from “A comparison of tools for teaching formal software verification”, by Ingo Feinerer and Gernot Salzer [16].

[8]`http://symbolaris.com/info/KeYmaera.html`

and maintenance of the Java programming language, before merging with Oracle Corporation in 2010. Moreover, it is also regarded as “the most popular parser generator for Java” [19]. The work of Mamas and Kontogiannis shows that the Java programming language itself can be parsed using JavaCC, going as far as translating Java classes into XML: a markup language they call “JavaML”.[19] This is further proven true due to the fact that the latest JavaCC version to date (version 5.0) is distributed with a grammar for Java 1.5 (located in javacc-5.0/examples/JavaGrammars). Finally, JavaCC also supports the building of Abstract Syntax Trees, through the aforementioned JJTree preprocessor.

SableCC

Another Java-based tool for generating compilers and interpreters is *SableCC*. Its authors, Etienne Gagnon and Laurie Hendren, published a rather short, yet concise paper that describes it, provides an overview of its features, and shows how a subset of the BASIC language can be parsed using it.[20] Some key issues are mentioned in said paper, including the fact that most modern compilers are generally multi-pass, and also the fact that many compilers work on Abstract Syntax Trees, built from the source code.

ANTLR

Also written in Java, *ANTLR* is a tool for the generation of LL(*) parsers. Both Parse Trees and Abstract Syntax Trees can be constructed using ANTLR [9], and in fact, the latter feature has been used in the work of Ning Li et al. to generate ASTs for the C programming language, for the purposes of detecting structural changes in code; ANTLR was praised in this work as a tool that “provides excellent support for tree construction, tree-walking, multi-languages”[21]. Wulf et al. have also opted to use ANTLR’s AST construction feature to perform the transformation from C# code to OMG’s Knowledge Discovery Metamodel, referring to the tool itself as “popular and matured”[22].

PLY

Python Lex-Yacc, otherwise known as *PLY* is, like Flex and Bison, an implementation of Lex and Yacc, though implemented in pure Python. A few projects use PLY, most notably *pycparser* [10]. No direct support is provided, by PLY, for constructing ASTs, but the official documentation describes how to (easily) do so [11].

Ricardo Martín Brualla has used PLY to translate C++ code into other programming languages, namely Pascal, Java and Python; his work’s main purpose was to evaluate execution times for these programming languages, in order to make adjustments to time limits, if needed[23]. Interestingly, PLY was also used for biological research purposes, in particular to write LipidXplorer [12], a “software that supports the quantitative characterization of complex lipidomes by interpreting large datasets of shotgun mass spectra”, as per the work of Herzog et al.[24]

[9]http://www.antlr.org/wiki/display/ANTLR3/Interfacing+AST+with+Java
[10]http://code.google.com/p/pycparser/
[11]http://www.dabeaz.com/ply/ply.html#ply_nn34
[12]https://wiki.mpi-cbg.de/wiki/lipidx/index.php/Main_Page

2.1.3 Network simulation

The ns series

ns-2 is a discrete-event simulator, designed to perform simulations at protocol level. This includes specifying whether a node shall send data using TCP or UDP, whether links between two given nodes are half-duplex or full-duplex, as well as the bandwidth and delay associated with those links. Furthermore, ns-2 contains a mobile wireless simulation model, for creating wireless scenarios. *OTcl*, an object-oriented version of Tcl, is ns-2's main programming language, and is the one that is used as the scripting language to model the simulation scenario. Additionally, C++ can be used to extend ns-2's functionality.

In fact, ns-2 is part of a series of discrete-event simulators which, at the present time, consists of ns-1, ns-2 and ns-3. Development for ns-1 and ns-2 has ceased, and so has maintenance for the former of which. However, ns-2 is still being maintained and has, in fact, been used fairly recently (September 2012) to simulate a scenario where a Denial of Service attack – named "Gray Hole Attack" – occurs, as per the work of Kanthe, Simunic and Prasad [25].

ns-3, unlike its predecessors, is still being actively developed and maintained, and drops the use of OTcl for scripting, using either C++ or Python for that purpose instead. However, as mentioned in the official tutorial for ns-3 [13], "ns-3 does not have all of the models that ns-2 currently has", hence it is sometimes preferable to use ns-2 instead.

GloMoSim

GloMoSim is "a scalable simulation environment for wireless and wired network systems" [14]. It is backed by *Parsec* [15], a simulation language based on the C programming language. The latest version of GloMoSim to date (2.03) is quite outdated, as it was released on December 19, 2001. Nevertheless, it was recently used for the purpose of comparing two routing related protocols by Sivaganesan and Venkatesan. [26]

GloMoSim is not limited to Network (Routing) protocols, however, as its library provides a vast range of protocols, associated with different layers. These include the well-known transport protocols TCP and UDP and even protocols at the application layer such as FTP, HTTP and Telnet. It is also possible to develop new protocols for GloMoSim, even though doing so requires some knowledge of Parsec.

TOSSIM

The popular, open-source operating system TinyOS includes its own simulator, *TOSSIM*, as part of its standard release. A wiki-based tutorial for TOSSIM is available [16]. In reality, TOSSIM should not be regarded as a simulator on its own, but rather a library that can be used for writing simulation code. This code can be written in either Python or C++, as

[13]`http://www.nsnam.org/docs/release/3.15/tutorial/ns-3-tutorial.pdf`
[14]`http://pcl.cs.ucla.edu/projects/glomosim/`
[15]`http://pcl.cs.ucla.edu/projects/parsec/`
[16]`http://docs.tinyos.net/tinywiki/index.php/TOSSIM`

TOSSIM provides interfaces for both.

Writing extensions for TOSSIM seems to be fairly common: Derhab et al. designed and implemented MOB-TOSSIM to include support for mobility in WSNs and WSANs [27]; back in 2008, Perla et al. ported the PowerTOSSIM extension to TinyOS 2.0, giving this port the name of *PowerTOSSIM-Z* [28]; Landsiedel et al. introduce *TimeTOSSIM* as an extension to TOSSIM associated with the concept of "Time Accurate Simulation" [29].

OMNeT++

Finally, network simulations may also be written in C++ using the *OMNeT++* framework (although OMNeT++ itself is not limited to this class of simulations). OMNeT++ is "free for academic and non-profit use", and is advertised as a "widely used platform in the global scientific community"[17]. OMNeT++ is distributed as a source code + IDE bundle; this bundle may also include MinGW, for source code compilation under Windows machines [18]. A PDF document with detailed installation instructions is also available [19]. Moreover, OMNeT++'s IDE is based on Eclipse, making the process of creating, compiling, and running projects relatively easy. OMNeT++ also possesses an extensive, documented API [20], a rich Tkenv graphical simulation environment, a tutorial for a simple messaging protocol (named "TicToc") [21], and comes with a wide range of examples/preset projects (currently 17), each with its own, detailed README file.

Like GloMoSim, OMNeT++ has also been used to perform simulation on routing protocols. Dräxler et al. used OMNeT++ for network-based power consumption optimization, choosing OMNeT++ due to "its modularity and the built-in features to dynamically set up a simulation" [30]. Comsa et al. used OMNeT++ to simulate the "Floyd-Warshall all-paths routing algorithm" [31].

OMNeT++ is not limited to routing and low-level network protocols, however. In fact, during the first semester of the academic year of 2012-2013, this document's author carried out an experiment in OMNeT++ related to a simple, application-level protocol which he wrote in PROMELA (and thus, accepted by Spin), as will be shown further in this document.

2.2 Related work

Generating source code from a formal model is an area of computing that has not been yet significantly explored. Nevertheless, some work has already been done in it. Iliasov has published a paper on the generation of source code from an Event-B model [32]. The work mentioned in that paper includes the implementation of a tool that takes an Event-B model as input and outputs either pseudo-code or a subset of Java with JML annotations. This tool is denominated

[17]http://www.omnetpp.org/

[18]http://www.omnetpp.org/omnetpp/cat_view/17-downloads/1-omnet-releases

[19]http://www.omnetpp.org/doc/omnetpp/InstallGuide.pdf

[20]http://www.omnetpp.org/doc/omnetpp/api/index.html

[21]http://www.omnetpp.org/doc/omnetpp/tictoc-tutorial/part1.html

B2H5, and is available for the Rodin Platform [22].

Perhaps of more interest is the fact that there is also research associated with NASA on the subject at hand. More specifically, there is a report by Lensink et al. on the generation of Java source code from a formal model whose specification is written in PVS (Prototype Verification System) [33]. The translation process from PVS to Java is not direct, however, as the report's authors mention that an intermediary language is used in-between both languages: the Why language [23]. The use of such language has one key benefit, as mentioned in the report: "in the future, the generator may be extended to support other functional and imperative programming languages".

Regarding KeY, the theorem prover that we mentioned before, some research work has addressed the problem of verifying C code. A variant of KeY for a subset of the C programming language has been developed, and is described in the work of Mürk, Larsson and Hähnle [35].

2.3 Choosing the adequate tools

From the various software tools presented in this chapter, we chose **Spin, JavaCC and OMNeT++** as the components related to PROMNeT++. For reasons presented in the next chapter, we have determined that they are the most adequate tools for reaching this project's goals.

[22]`http://iliasov.org/b2h5/`

[23]Please note that the original Why platform, although still maintained, is no longer actively developed. It is available at `http://why.lri.fr/`

Chapter 3

Objectives and methodology

As mentioned in Chapter 1, the main objective of this work was to ascertain the possibility of generating runnable source code from models pertaining to the domain of round-based consensus protocols. This was indeed proven to be possible, as PROMNeT++ ships with two distinct round-based consensus protocols, and generates C++ code that can be successfully compiled and simulated via OMNeT++ and its IDE. In fact, a tutorial on how to do this is available on PROMNeT++'s Wiki, provided by Google Code[1].

It was also in this project's interest to determine the generated source code's level of quality and reusability. First and foremost, there was the need to guarantee that the generated code is an implementation of the original model. In other words, the generated source code must be guaranteed to correspond to the formal description provided by the model. As described later in this chapter, after translating a model, we simulated the resulting code with the same set of scenarios (test cases) as the original model, in order to ensure that the output is the same.

Regarding the problem of ensuring the code's reusability, our approach was to design the translation process so as to avoid generating unnecessary or superfluous code. Since the PROMELA language has many similarities with C, we chose to maintain the structure of blocks of code (for instance, "if" statements in PROMELA are translated into "if" statements in C++, "inlines" in PROMELA are translated into C++ functions). The intention with this approach was to ensure that the generated code has roughly the same complexity as the original PROMELA model.

To validate the aforementioned design choices, we reviewed the generated code manually, in order to ensure that the result did not add unnecessary complexity, and we computed the McCabe complexity for the various routines in order to make sure that the numbers were within the expected values. Given that it would also be of interest to researchers to run the generated code in real systems, Raul Barbosa (this internship's supervisor) held an informal meeting, with researchers from another institution, who found no obstacles related to the generated code itself that would prevent this from running in a real system.

[1]`https://code.google.com/p/promnetpp/wiki/CommandLineWorkflow1`

3.1 Chosen software tools

3.1.1 Model checking

Spin was already defined as the model checking tool of choice, even before the beginning of this project, and was thus used. Firstly, Spin is free and open-source unlike, for instance, Cadence SMV. Furthermore, as its official website states, Spin "was developed at Bell Labs in the original Unix group of the Computing Sciences Research Center, starting in 1980", making it highly mature and, as shown in chapter 1, highly performant as well. More importantly, this internship's supervisor, Raul Barbosa, has worked extensively with Spin before, and thus possesses the know-how on the model checking tool; he was, in fact, able to supervise the project's development, and able to provide additional knowledge on Spin.

3.1.2 Parsing/language recognition

The parsing tools listed in the previous chapter are all very mature, despite the fact that, when PROMNeT++ was being designed/developed, the latest stable release of JavaCC dated back to 2009, as well as the latest revision for PLY (according to its Google Code repository [2]).

Our goal was to implement PROMNeT++ in the Java programming language, despite the fact that non-Java alternatives such as Flex + Bison and PLY would be perfectly suitable for our project's needs. We anticipated the possibility of integrating our translator with the popular Eclipse IDE (in which OMNeT++ is based on, OMNeT++ being another tool of choice, as described below), and given that plug-in development for Eclipse is done in Java, choosing it as a programming language facilitated the integration between PROMNeT++ and Eclipse; in fact, a plug-in for PROMNeT++ was indeed developed, along with the tool itself, and is available at the project's download page in Google Code [3], under the name "PROMNeT++ Plugin, version 0.0.2". Using Java also provides the benefit of passing on the project to a wide selection of future Internship/Thesis students over at the Department of Informatics Engineering, who will likely have the know-how in Java to maintain and expand the project.

Amongst the Java-based alternatives, JavaCC was determined to be the most adequate for the job. JavaCC has a large user community, which can be reached through several mailing lists, most notably the "Users" mailing list[4], which can be resorted to should there be a need for help using the tool in general. Several JavaCC books are also available, including Tom Copeland's "Generating Parsers with JavaCC, 2nd edition", available at `http://www.generatingparserswithjavacc.com/`, complete with a large collection of code examples available for download, free of charge. Additional books related to JavaCC are also readily available at the Department of Informatics Engineering's library, to which the author had access, if needed.

Given the above benefits, **JavaCC was used as one of the building blocks for PROMNeT++**. Using JJTree, which is bundled with JavaCC, a PROMELA parser[5] was constructed,

[2] `http://code.google.com/p/ply/source/list`

[3] `https://code.google.com/p/promnetpp/downloads/list`

[4] `http://java.net/projects/javacc/lists/users/archive`

[5] This is not a fully fledged PROMELA parser, and it most likely does not parse the entirety of PROMELA;

whose code is available at `https://code.google.com/p/promnetpp/source/browse/trunk/promnetpp/src/com/googlecode/promnetpp/parsing/Parser.jjt`.

3.1.3 Network simulation

ns-2 and its successor, ns-3, although very popular in the area of network simulation, weren't suitable for this project's purposes, as they were designed for protocols *below the application layer* such as TCP and UDP. It is possible to extend their functionality by writing modules in C++; however, there are alternatives which don't require this.

GloMoSim, despite being able to (and it does) simulate protocols at the application layer, requires some knowledge in Parsec in order to write new protocols of the same class. TOSSIM, on the other hand, provides both Python and C++ interfaces for writing simulations. Unfortunately, TOSSIM restricted to the TinyOS platform.

Out of all the listed alternatives, **OMNeT++ came out on top**. First and foremost, OMNeT++ itself is very easy to deploy, especially in Windows platforms, as one of its distributions comes pre-packaged with MinGW. Step by step instructions are available in the official installation guide[6]. In fact, compiling OMNeT++ under Windows requires very little effort. All it takes is to open the "mingwenv.cmd" file that comes with the distribution, and execute the following commands:

```
./configure
make
```

After executing the above commands in Windows, OMNeT++ became fully ready to use. Its Eclipse-based IDE can be opened by simply issuing the "omnetpp" command.

Furthermore, OMNeT++ uses C++ code for simulation. Worth mentioning here is the fact that C++ is ranked amongst the top 5 programming languages of 2012, according to TIOBE [7]. Also important is the fact that OMNeT++ possesses a graphical simulation environment, which is rather flexible. In particular, it is possible to adjust the speed of a particular simulation, as well as customizing the icons for each node in the network. Additionally, OMNeT++'s IDE is based on Eclipse, making it easier to create and run simulation projects, and featuring code completion for its C++ API. Additional features include a message compiler ("opp_msgc") and a makefile generator ("opp_makemake").

To sum up, OMNeT++ has a set of advantages to the other network simulation alternatives that made it the strongest candidate for a tool of choice. Due to this set of advantages, **OMNeT++ was chosen as the software for network simulation**.

nonetheless, it works perfectly for the protocols listed in this document. Also, it extends PROMELA's grammar by including comments in the specification, which are necessary for annotations in PROMELA code.

[6]`http://www.omnetpp.org/doc/omnetpp/InstallGuide.pdf`

[7]`http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`

3.2 PROMNeT++'s main workflow

Given the choice of tools above, a software stack was assembled using them. The figure below is a diagram that illustrates the corresponding high-level specification.

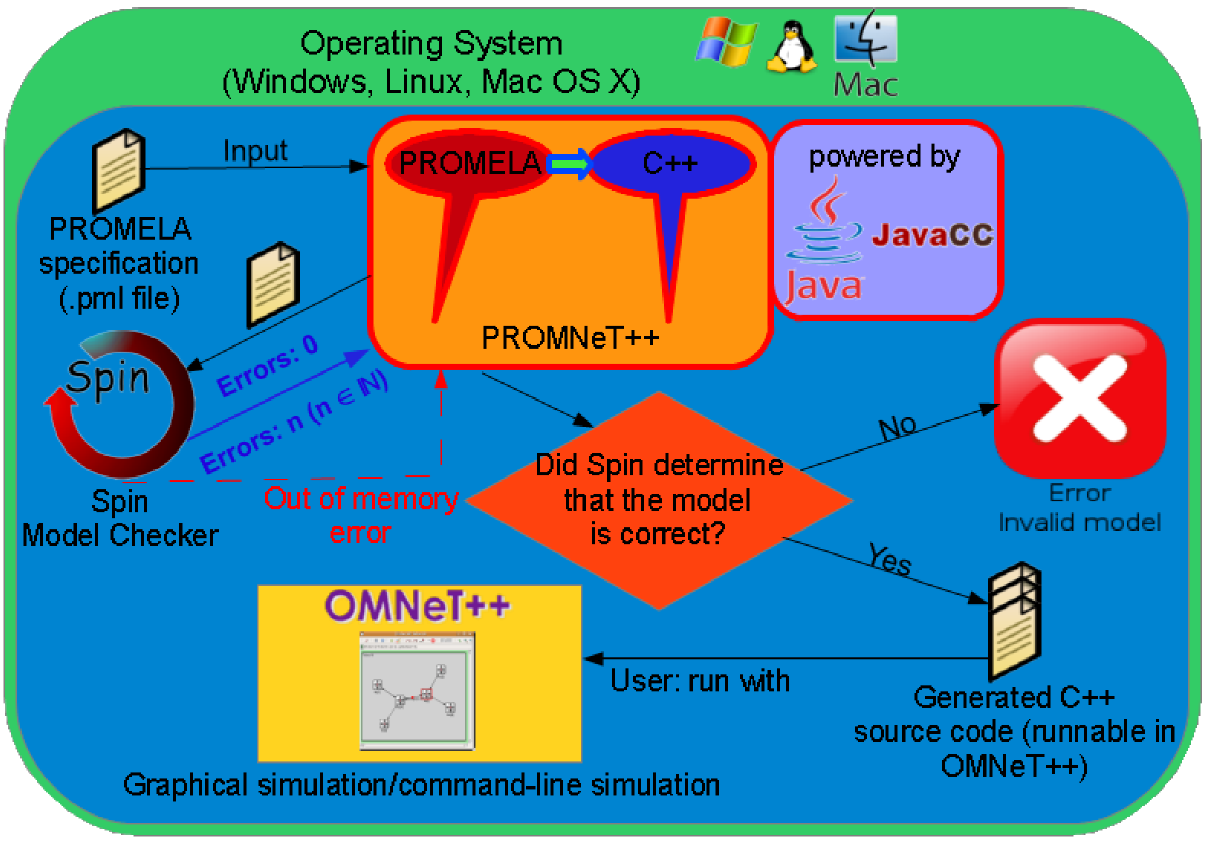

Operating System
(Windows, Linux, Mac OS X)
Mac
Input
PROMELA
C++
PROMNeT++
powered by
JavaCC
Java
PROMELA specification (.pml file)
Spin
Errors: 0
Errors: n (n ∈ ℕ)
Spin Model Checker
Out of memory error
Did Spin determine that the model is correct?
No
Error
Invalid model
Yes
OMNeT++
User: run with
Generated C++ source code (runnable in OMNeT++)
Graphical simulation/command-line simulation

Figure 3.1: The software stack, at a glance

Roughly put, PROMNeT++ takes a PROMELA model as input (a text file, typically with a .pml extension), invoke Spin for the purposes of checking the correctness of the specification, and (if it is correct[8]) generate a set of files containing, for the most part, C++ code (some will contain code specific to OMNeT++, such as the network's layout).

3.3 Software development methodology

3.3.1 Development tools

IDE

To facilitate the whole development process of the translator tool, a Java Integrated Development Environment (IDE) should be used. Several choices for Java IDEs exist, the most notable being NetBeans, Eclipse and IntelliJ IDEA. Out of these, this work's author (Miguel Martins),

[8]Or if the user has chosen to skip the verification procedure, which PROMNeT++ does by default.

has much more experience with NetBeans than the remaining alternatives, and uses its code completion feature quite extensively. Moreover, NetBeans integrates well with JUnit, a popular unit testing framework for Java, and the one that was used for this purpose, as will be described later. **NetBeans is, therefore, the IDE of choice**.

Unit testing framework

JUnit is chosen here for unit testing purposes. When installing NetBeans, the user (developer) is prompted to install JUnit alongside it. Once installed, the user may easily create unit tests and/or test suites from within the IDE; the user may also run the tests they have created by simply right-clicking on the respective project and choosing the "Test" option (or by using the *Alt+F6* keyboard shortcut).

3.3.2 Development platforms/environments

SVN repository and tools

Since the beginning of the project, an SVN repository had been assigned to both the project's author and its supervisor. This SVN repository is provided by University of Coimbra's Department of Informatics Engineering's Helpdesk, and is (privately) accessible via web at `https://svn.dei.uc.pt/usvn/login/`. PROMNeT++'s code was, at first, committed to said repository, before migrating to its current repository, at Google Code.[9]

Regarding software tools to manage said SVN repository, **TortoiseSVN** was used by the project's author. Firstly, it must be taken into account that the author frequently uses Windows (usually, Windows 7) for development (as well as for everyday tasks), and has used it extensively in the past. TortoiseSVN integrates well with Windows Explorer, bringing a Context Menu for folders marked as SVN repositories, which provides options for several SVN tasks (commands), including options for updating and committing, creating branches, locking and unlocking the repository (useful for binary files), and reverting to previous revisions if needed. Lastly, if more flexibility is needed, TortoiseSVN still offers the possibility of using the respective SVN command-line tools.

3.3.3 Quality Assurance Plan

Code style and conventions

Despite the fact that only a single person (the author) coded PROMNeT++, it is still of importance to keep a consistent coding style, to promote the code's maintainability. The quality assurance plan below was established during the internship's first semester for this very purpose; the guidelines are as follows:

1. Indentation shall be done using *spaces only*; using tabs will be avoided, unless strictly necessary.
   (a) Four (4) spaces per indentation level.

[9]`http://promnetpp.googlecode.com/svn/trunk/`

   (b) Many IDEs (NetBeans included) already support the conversion from tabs to spaces.

2. No line (whether it's part of a comment or actual code) should exceed 80 characters (excluding line terminators, since they're not visible) unless strictly necessary.
   (a) Line breaking can and should be used extensively (but not excessively) for this purpose.
3. Variable names and identifiers shall be kept short, yet concise.
   (a) More objectively, each variable name/identifier should be as descriptive as possible, without going over 30 characters in length.
   (b) As per Oracle's Code Conventions, class names shall use upper camel case (example: "MyClass" instead of "Myclass" or "myclass"), while variable names, function identifiers and function parameters shall use lower camel case (examples: "int lineNumber", "void someFunction(int someIntegerArgument)".
   (c) Acronyms in identifiers should not have their case altered (example: "ASTFactory" instead of "AstFactory").
4. Javadoc comments (documentation comments) and respective annotations should be used extensively, in particular for classes and methods.

Product development cycle

Standard SVN procedure for development, for text files, is as follows:

1. SVN update.
2. Make changes to the code.
3. SVN update, followed by SVN commit when finished.

For binary files:

1. SVN update.
2. SVN lock.
3. Make changes to the files in question.
4. SVN commit.
5. SVN unlock.

The planned development cycle roughly consisted in the creation of artifacts (text/binary files), committing the corresponding changes to the SVN repository as described above, and repeating the whole process until either a milestone or the solution has been reached. It was also proposed, during the first semester, that for each new feature, a set of tests would be written (using JUnit) to enhance the feature's correctness. Unfortunately, this proposition ended up not taking effect.

3.4 Round-based model

Round-based consensus protocols essentially consist of a set of nodes which repeatedly call a very specific list of routines, in a perpetual loop. The pseudocode below illustrates this loop.

```
1: loop
2:     begin_round()
3:     compute_message(my_message)
4:     send_to_all(my_message)
5:     wait_to_receive()
6:     state_transition()
7:     end_round()
8: end loop
```

This structure was proposed by Raul Barbosa himself[10], in an attempt to unify all round-based consensus protocols in a simple, generic pattern. There are six routines in total, as shown above. *begin_round* and *end_round* signal the beginning and the end of a particular round, respectively. Once a particular round has begun, each process computes its own message via *compute_message*, before sending it to all processes (including itself) via *send_to_all*. Processes must then wait for all messages to arrive by calling *wait_to_receive*. Finally, before ending a particular round, processes must update their own state via *state_transition*, according to the norms of the particular protocol they're following. This structure is described in greater detail in the next chapter.

3.5 PROMNeT++'s template system

PROMNeT++ does not fully generate C++ code on its own; instead, it uses pre-written C++ templates and fills in portions of those templates accordingly. "Template", here, is defined as a text file with one or more template parameters; a template parameter is a string such as "{0}", "{1}", "{2}", and so forth. In other words, a template parameter is simply a string that contains the representation of an integer greater than zero, enclosed in curly brackets, like so:

$$\{n\}, n \geq 0$$

Template parameters are replaced, at some point during translation, with real C++ code, and can thus be regarded as placeholders for it; replacement is done via Java's *MessageFormat* class [11]. Files that contain template parameters have the entirety of their contents read to memory, as *String* objects, then have its template parameters replaced, and are finally written back to disk, as output. A simple example of this is the *types.h* file, which holds PROMELA's user types:

```
#ifndef TYPES_H_
#define TYPES_H_

#include "global_definitions.h"

typedef unsigned char byte;

{0}

```

[10] A minor variant of it can be seen in a draft document hosted at Chalmers University of Technology's Computer Science and Engineering website. Said document can be found here: http://www.cse.chalmers.se/~johan/publications/TR2013.pdf

[11] http://docs.oracle.com/javase/6/docs/api/java/text/MessageFormat.html

```
#endif /* TYPES_H */
```

In this case, there is only a single template parameter ("{0}"). PROMNeT++ replaces this parameter with *typedef struct* statements derived from PROMELA's own *typedef*. For instance, in the PROMELA model for the OneThirdRule protocol, the following typedefs are present:

```
/* NOTE: This is PROMELA code, not C/C++ code */
typedef message {
    byte value;
}

typedef process_state {
    bool received_message[NUMBER_OF_PROCESSES];
    byte received_message_count;

    byte local_value;
    byte decision_value;
    byte values[NUMBER_OF_PROCESSES]
}
```

PROMNeT++ internally maps each of PROMELA's *typedef* statements into a C/C++ *typedef struct* statement, and replaces the above template parameter with the result. The resulting output is as follows:

```
#ifndef TYPES_H_
#define TYPES_H_

#include "global_definitions.h"

typedef unsigned char byte;

typedef struct {
    byte value;
} message_t;

typedef struct {
    bool received_message[NUMBER_OF_PROCESSES];
    byte received_message_count;
    byte local_value;
    byte decision_value;
    byte values[NUMBER_OF_PROCESSES];
} process_state;

#endif /* TYPES_H */
```

3.6 Work/experimentation methodology

To test the accuracy of PROMNeT++'s translation process and produce verifiable results, our PROMELA models need to follow a very specific set of rules. PROMELA models are, by nature, non-deterministic, meaning that whenever a point of non-determinism is reached, Spin (when running a random simulation) uses its own pseudo-RNG to decide which path it should take. Given that the translation process has no knowledge of said pseudo-RNG, it was necessary to embed, within the PROMELA model itself, a simple linear congruential generator, and then rewrite any points of non-determinism to use said generator, thereby bypassing Spin's pseudo-RNG.

3.6.1 Eliminating non-determinism in PROMELA: pseudo-RNG substitution

As mentioned before, most PROMELA models are inherently non-deterministic. Consider the following piece of PROMELA code (simplified from the OneThirdRule protocol) as an example:

```
#define NUMBER_OF_PROCESSES 3

typedef process_state {
    bool received_message[NUMBER_OF_PROCESSES];
    byte received_message_count;

    byte local_value;
    byte decision_value;
    byte values[NUMBER_OF_PROCESSES]
}

process_state state[NUMBER_OF_PROCESSES];

inline system_init() {
    j = 1;
    for(i : 0..(NUMBER_OF_PROCESSES-1)) {
        state[i].local_value = j;
        /*
        This is a non-deterministic if statement; there's a 50%
        chance it will choose the "j++" statement, and a 50% chance
        it will choose the "skip" statement ("skip" statements do
        absolutely nothing).
        */
        if
        :: j++
        :: skip
        fi;
        printf("MSC: P%d has initial value x=%d\n", i+1,
            state[i].local_value)
    }
}

init {
    int i, j;
    system_init()
}
```

In the above scenario, each process inherits the value of the j variable, which has an initial value of 1. When simulating this code via Spin, Spin has to determine whether it will execute the first or the second set of statements (delimited by "::") that belong to their corresponding "if" block (delimited by "if ... fi"). Since both sets ("j++" and "skip") are executable[12], they have each a 50% probability of being executed. Effectively, this means that it is possible that the j variable might not be incremented at all, as it's also possible that it could be incremented just once. Possible outputs for this scenario include:

```
C:\spin>spin temp.pml
      MSC: P1 has initial value x=1
      MSC: P2 has initial value x=1
      MSC: P3 has initial value x=1
1 process created

C:\spin>spin temp.pml
      MSC: P1 has initial value x=1
      MSC: P2 has initial value x=2
      MSC: P3 has initial value x=2
1 process created

C:\spin>spin temp.pml
      MSC: P1 has initial value x=1
      MSC: P2 has initial value x=2
      MSC: P3 has initial value x=3
1 process created
```

A possible C++ implementation for this scenario consists in using OMNeT++'s Mersenne Twister, and in particular, OMNeT++'s *intrand(n)* function, which produces a random integer in the interval $[0, n[$. Since there's only 2 sets of statements ("j++" and "skip"), one could make n equal to 2, thus obtaining either 0 or 1 as a possible value for *intrand(n)*; if 0 is obtained, then execute "j++"; if 1 is obtained instead, execute "skip" (i.e. do nothing). In C++ code, this would be as follows:

```
void system_init() {
    j = 1;
    for(i = 0; i <= (NUMBER_OF_PROCESSES-1); ++i) {
        state[i].local_value = j;
        int decision = intrand(2);
        if (decision == 0) {
            j++;
        } else if (decision == 1) {
            //skip
            //(note that there is no "skip" statement in C++)
            //(this block does absolutely nothing, just like
            // "skip")
        }
        printf("MSC: P%d has initial value x=%d\n", i+1,
            state[i].local_value)
    }
```

[12]More on the concept of "executability" at http://spinroot.com/spin/Man/Manual.html

```
}
```

While this implementation works perfectly, it poses a problem: we are, in fact, using OMNeT++'s Mersenne Twister via *intrand(n)*, while Spin uses its own pseudo-RNG. Thus, using this approach would make it unfeasible to carry out experiments such as comparing Spin's output to OMNeT++'s, which are later shown in this document for the purpose of determining if the translation process produces a correct implementation of the original PROMELA model.

We solve this problem by embedding a user-defined pseudo-RNG in the PROMELA model, and rewriting it to become fully deterministic. We opted for a linear congruential generator, due to its simplicity in implementation and use. The Wikipedia article for linear congruential generators lists over 10 distinct sets of LCG parameters[13]. Given that we were looking for simplicity over randomness, we opted for an LCG known as *MINSTD*, used in Apple's CarbonLib. The recurrence relation for this LCG is as follows:

$$X_{n+1} = (X_n \times 16807) \bmod 2147483647$$

Like most pseudo-RNGs in existence, we need to declare a variable that will initially hold the seed and the pseudo-random value at any later point in time. Secondly, we also need a routine that can be repeatedly called to get the next number in the pseudo-RNG sequence. For our convenience, we also created a routine that converts the aforementioned pseudo-random value into a boolean value; this helps with binary decision *if* statements such as the one described above. All of these are implemented in PROMELA as follows:

```
/* Please note that, while this code also works for C/C++, this is
actually embedded into the PROMELA model. */
/* Random number generation */

int rnd = 1234; /* 1234 is the seed for our LCG; we can change it
if we so desire. */
#define next(r) (r * 16807) % 2147483647
#define boolean(r) ((r >> 30) & 1)
```

Using this technique, we can rewrite our *system_init* routine as follows:

```
inline system_init() {
    j = 1;
    for(i : 0..(NUMBER_OF_PROCESSES-1)) {
        state[i].local_value = j;
        rnd = next(rnd); /* "rnd" refers to the global variable
            that we previously declared */

        if
        :: boolean(rnd) -> j++
        :: else -> skip
        fi;

        printf("MSC: P%d has initial value x=%d\n", i+1,
            state[i].local_value)
```

[13]https://en.wikipedia.org/wiki/Linear_congruential_generator#Parameters_in_common_use

```
    }
}
```

By rewriting our PROMELA models so that any binary decision points (such as the one above) are actually dependant on the above custom pseudo-RNG, we are effectively making them completely deterministic[14]; we are now in complete control of the execution of our models, and we're only required to change our LCG's seed if a different execution is desired. Additionally, PROMNeT++ parses and translates the definitions for our custom pseudo-RNG.

3.6.2 Obtaining results

Using what has been described so far in the current section (3.4), we can perform a very simple, yet necessary experiment. To ensure that PROMNeT++ generates source code that is faithful to the original PROMELA model (i.e. the generated code is a valid implementation of the model), we can choose a seed, and compare the output from simulating the model via Spin to the output that results from running a simulation in OMNeT++. The latter, of course, implies that we use PROMNeT++ to perform a translation from PROMELA to C++, then compile the generated source code using OMNeT++'s compiler tools into an executable file, and finally run this executable file to obtain the output from OMNeT++'s simulation. If there is a match between both outputs ("PROMELA/Spin" and "C++/OMNeT++"), then we claim that the generated source code is a valid implementation of the model.

As an example, consider Spin's output for the OneThirdRule protocol that's distributed with PROMNeT++, using the value 1234 as the LCG's seed:

```
MSC: P1 has initial value x=1
MSC: P2 has initial value x=1
MSC: P3 has initial value x=1
MSC: new round, id=1
rnd=892703006
MSC: P1 decides 1 on round 1
MSC: new round, id=2
rnd=1896731518
MSC: P3 decides 1 on round 2
MSC: P2 decides 1 on round 2
MSC: P1 decides 1 on round 2
MSC: new round, id=3
rnd=1896731518
MSC: P3 decides 1 on round 3
MSC: P2 decides 1 on round 3
MSC: P1 decides 1 on round 3
MSC: new round, id=4
rnd=1896731518
MSC: P3 decides 1 on round 4
MSC: P2 decides 1 on round 4
```

[14]There is actually one other source of determinism in PROMELA; at any given point in time, any given process may be switched out for another, akin to how a multi-threaded application would swap out one thread for another. This source of non-determinism is irrelevant, however, as it only changes the order of the *printf* statements per process.

```
MSC: P1 decides 1 on round 4
MSC: new round, id=5
rnd=1896731518
MSC: P3 decides 1 on round 5
MSC: P2 decides 1 on round 5
MSC: P1 decides 1 on round 5

(output truncated for brevity...)
```

Now, consider the output that is obtained if we translate the model via PROMNeT++, then perform an OMNeT++ simulation:

```
MSC: P1 has initial value x=1
MSC: P2 has initial value x=1
MSC: P3 has initial value x=1
MSC: new round, id=1
rnd=892703006
MSC: P1 decides 1 on round 1
MSC: new round, id=2
rnd=1896731518
MSC: P2 decides 1 on round 2
MSC: P1 decides 1 on round 2
MSC: P3 decides 1 on round 2
MSC: new round, id=3
rnd=1896731518
MSC: P2 decides 1 on round 3
MSC: P1 decides 1 on round 3
MSC: P3 decides 1 on round 3
MSC: new round, id=4
rnd=1896731518
MSC: P2 decides 1 on round 4
MSC: P1 decides 1 on round 4
MSC: P3 decides 1 on round 4
MSC: new round, id=5
rnd=1896731518
MSC: P2 decides 1 on round 5
MSC: P1 decides 1 on round 5
MSC: P3 decides 1 on round 5

(output truncated for brevity...)
```

We can clearly see that both outputs are practically identical, only differing in the order that processes P1, P2 and P3 are executed. For example, in round 5, P3 is the first process to decide the value 1 in Spin, while being the last process to decide the same value under OMNeT++'s simulation. Process execution order, however, is unimportant for our experiments, and is merely the result of Spin and OMNeT++ scheduling processes differently.

Later on in this document, we repeat this experiment for 29 more seeds, and observe that the outputs from both Spin and OMNeT++ are, in fact, identical.

Chapter 4

Requirements analysis

4.1 Stakeholder identification

The stakeholders for this project consist of, essentially, every person who is interested in the formal verification of distributed algorithms and wants to be able to execute runnable, ready to be used code in real systems. An example of two such people would be Tatsuhiro Tsuchiya and André Schiper, as they have published a paper on the verification of consensus algorithms through (bounded) model checking [34].

Raul Barbosa, this internship's supervisor, acts in this project as a "proxy" for that large group of stakeholders; in a way, he represents this group, as he has also expressed his interest in this project's area of computing. In fact, he, along with Johan Karlsson, have worked on the subject at hand, as per the work in their article previously cited in this document. [5]

4.2 User stories

4.2.1 Rationale

As with most software projects, requirements are gathered here by derivation from a set of scenarios and/or interactions. Given that there aren't many ways for a user to interact with PROMNeT++, a few user stories will be presented in this section. The construct to be used for the user stories is described in Kelly Waters' "All About Agile" blog's entry on "User Stories"[1], and is presented as follows:

As a [user role], I want to [goal], so I can [reason].

4.2.2 Specification

1. As a user of the PROMELA to C++ translator application and developer for critical systems, I want to be able to invoke the tool from the command line, passing only the name or path to a PROMELA (.pml) specification file as the sole argument, so I can let

[1] `http://www.allaboutagile.com/user-stories/`

the tool choose any other parameters for me and reuse the generated C++ source code (if any).

2. As a user that must work with a machine with limited memory, I want to be able to invoke the tool from the command line, passing not only a PROMELA file as an argument, but also an indication to use a memory saving profile, so I can reuse the generated C++ source code (if any) while avoiding the need to upgrade my machine's RAM.
3. As a user, I want to be able to invoke the tool from the command line, passing both a PROMELA file and a configuration file as arguments, so I can adapt the tool's behavior to machines with different hardware configurations (namely CPU and RAM).
4. As an Eclipse IDE user/developer, I want to be able to right-click on my project's PROMELA file and choose the option to translate it to C++, so I can avoid interacting with the tool's command line interface while benefiting from the comfort provided by a graphical environment.
5. As a cross-platform developer, I want to be able to invoke the tool under a Windows, Linux, or OS X environment, so I can reuse the generated C++ code (if any) without worrying about operating system restrictions.
6. As a user of the PROMELA to C++ translator application, I want to be able to visualize the interaction and/or results of my algorithm/protocol, so I can further ensure that my system works as I intend it to.

4.3 Requirements listing

The technical and functional requirements listed here were gathered and established during the first semester of the internship. Some of the requirements are labelled as "optional", meaning that their completion is not strictly necessary to fulfill the internship's objectives. Finally, any given requirement is either labelled as "met" or "not met", depending on whether its fulfillment was reached or not.

4.3.1 Technical requirements

Requirement	**Optional?**	**Met?**	**Notes**
The tool must be cross-platform.	No	Yes	It must, at least, be runnable under all major operating systems (Windows, Linux, OS X).
The tool must verify the model specification, using Spin, prior to translation to C++ code.	No	Yes	PROMNeT++ was indeed programmed to verify its input file, a PROMELA model, prior to its translation phase, and report any errors within said model. This feature is turned off by default, but can be turned on by editing PROMNeT++'s main configuration file.
The tool must be written in pure Java.	No	Yes	
The tool must generate C++ source code, ready to be executed in OMNeT++.	No	Yes	

4.3.2 Functional requirements

Requirement	**Optional?**	**Met?**	**Notes**
The tool must give its users the option to specify a single command-line argument, which is the name or path to a PROMELA specification file.	No	Yes	PROMNeT++ can be executed by performing a simple "java -enableassertions -jar promnetpp.jar mymodel.pml" command. Aside from Java's command-line switches, and the path to PROMNeT++'s JAR file, "mymodel.pml" is the sole command-line argument.
The tool must allow its users to create and use profiles, to tweak the tool's behavior to match their needs.	Yes	No	It is only compulsory that the tool provides a single command-line argument, as mentioned in the above requirement.
A plugin for the Eclipse IDE must be written, so that users can, for instance, translate a PROMELA specification directly from OMNeT++'s IDE.	Yes	Yes	An Eclipse plugin for PROMNeT++ does indeed exist, and is available at the project's "Downloads" page at Google Code; source code for this plugin is also available.
A GUI wrapper must be written for the command-line tool.	Yes	No	

4.4 Detailed requirements for the translation process

It had been planned that PROMNeT++ would translate PROMELA models that follow a very specific structure. In this section, we describe this structure in more detail, and promptly show PROMNeT++'s annotations.

4.4.1 Round-based model

Round-based consensus protocols have a well-defined structure in PROMELA. There are two types of nodes in the whole system: the *init* node, and the *Process* node. In PROMELA, both nodes are actually designated as "processes", and can also thus be referred to as the *init* process, and the *Process* process, respectively:

- The *init* process serves as the system's coordinator, and is responsible for starting all other processes and keeping them synchronized between rounds; there is only one instance of this process in the whole system.
- The *Process* process communicates, via message passing, with other processes of its kind, and is responsible for performing all necessary computations in order to reach a decision, as any round-based consensus protocol mandates; there is always more than one instance of this process in the whole system.

The *init* process is defined, in PROMELA, as follows:

```
init {
    /* Variable declarations go here. Below is a typical example.
    */
    byte i, j;
    bool synchronous = false;

    /* system_init MUST be init's first called routine */
    system_init();

    /* Required, so that the actual processes run */
    atomic {
        for(i : 1..(NUMBER_OF_PROCESSES)) {
            run Process()
        }
    }

    /* Process synchronization between rounds (Raul Barbosa's
    implementation via token passing; "token" is a global
    variable, of the byte type) */
    do
    :: (token == 0);
        system_every_round();
        token = NUMBER_OF_PROCESSES;
        (token == 0);
        token = NUMBER_OF_PROCESSES
    od
}
```

The *Process* process, on the other hand, is coded as shown below:

```
proctype Process() {
    /* Variable declarations go here. Below is a typical example.
    */
    message _message;
    byte i, j;

    /* Processes are STRICTLY REQUIRED to call the routines below
    in the order they're shown: begin_round, compute_message,
    send_to_all, wait_to_receive, state_transition, end_round */

    do
    :: begin_round();
        compute_message(_message);
        send_to_all(_message);
        wait_to_receive();
        state_transition();
        end_round()
    od
}
```

Finally, the overall structure for a round-based consensus protocol in PROMELA is established like so:

```
/* @UsesTemplate(name="round_based_protocol_generic") */
/* @TemplateParameter(name="numberOfParticipants") */
#define NUMBER_OF_PROCESSES 3 /* Can be any value greater than 1
*/

inline compute_message(_message) {
    /* Code for compute_message */
}

inline state_transition() {
    /* Code for state_transition */
}

inline system_init() {
    /* Code for system_init */
}

inline system_every_round() {
    /* Code for system_every_round */
}

/* @BeginTemplateBlock(name="generic_part") */

inline begin_round() {
    /* Code for begin_round */
}
```

```

inline end_round() {
    /* Code for end_round*/
}

inline send_to_all(_message) {
    /* Code for send_to_all */
}

inline wait_to_receive() {
    /* Code for wait_to_receive */
}

inline receive(_message, id) {
    /* Code for receive */
}

proctype Process() {
    /* See above */
}

init {
    /* See above */
}
/* @EndTemplateBlock */
```

A few important remarks about the protocol structure

As shown in the overall structure above, round-based consensus protocols are divided into a into two parts: a *specific part* and a *generic part*. The generic part is delimited by the *@BeginTemplateBlock(name="generic_part")* and *@EndTemplateBlock* annotated comments, which are used internally by PROMNeT++; any code not enclosed within the generic part is considered to be part of the specific part. The rationale for having these two parts is quite simple: **any code contained within the generic part remains constant across all protocols, with very few (if any) variations.** Thus, PROMNeT++ does not translate any such code, and can (and does) instead re-use pre-written C++ code for all generic elements.

The annotated comments used in the PROMELA models are described later in this chapter, in more detail.

Protocol routines and their implementations

- **begin_round** Generic routine. Ensures, via synchronization, that all processes are ready to start a new round. Called from *Process*.

```
inline begin_round() {
    (token == _pid)
}
```

- **end_round** Generic routine. Performs all the required actions to declare that a particular round is over. This includes any action required so that all processes can be synchronized before calling *begin_round* once again. Called from *Process*.

```
inline end_round() {
    token--
}
```

- **send_to_all** Generic routine. Sends the process' own message to all processes in the system (including itself, but excluding the init process). Called from *Process*.

```
inline send_to_all(_message) {
    messages[_pid-1].value = _message.value
}
```

- **wait_to_receive** Generic routine. Waits for all messages to be sent, before calling *receive* next. Called from *Process*.

```
inline wait_to_receive() {
    token--;
    (token == _pid)
}
```

- **receive** Generic routine. Performs a "receive" operation, which typically means taking a message from some queue/array/data structure, and storing it in the process' local state. Called from *Process*.

```
inline receive(_message, id) {
    _message.value = messages[id].value
}
```

- **compute_message** Protocol-specific routine. During each round, each process calls this routine to compute (i.e. alter the contents of, whenever necessary) its own message object. Called from *Process*.

```
inline compute_message() {
    /* Protocol-specific; no implementation is shown here */
}
```

- **state_transition** Protocol-specific routine. Performs the main computation(s) before ending the current round, usually by taking messages from the other processes and comparing them to its own; typically calls *receive* in a conditional loop; any consensus-related decisions are made in this function too. Called from *Process*.

```
inline state_transition() {
    /* Protocol-specific; no implementation is shown here */
}
```

- **system_init** Protocol-specific routine. Initializes the system's state, typically by setting individual parameters in process' states via pseudo-RNG. Called from *init*.

```
inline system_init() {
    /* Protocol-specific; no implementation is shown here */
}
```

- **system_every_round** Protocol-specific routine. Executes once before each round. Used to set the system's state for a particular round. Called from *init*.

```
inline system_every_round() {
    /* Protocol-specific; no implementation is shown here */
}
```

4.4.2 PROMNeT++'s annotations

To meet the technical requirements above and construct a powerful software tool, PROMNeT++ was built with support for annotated comments. An annotated comment is defined as a C-style comment whose contents begin with the @ symbol. These are inspired by Java annotations, and in particular, annotations used in the Java Persistence API[2].

There are currently four annotations in total, extensively described in PROMNeT++'s wiki[3]. Below is a concise summary of what each of them does.

- **@UsesTemplate(name="round_based_protocol_generic")** indicates PROMNeT++ that the input file is to be treated as a model for a round-based consensus protocol. *It must be the very first annotation in the PROMELA model.*
- **@TemplateParameter(name="numberOfParticipants")** indicates PROMNeT++ that the next *#define* directive is the number of participants in the round-based consensus protocol (excluding the *init* process). PROMNeT++ uses this numeric value to generate the appropriate network layout file for OMNeT++.
- **@BeginTemplateBlock(name="generic_part")** indicates PROMNeT++ that the model's generic part begins at this point. It must be used in conjunction with @EndTemplateBlock.
- **@EndTemplateBlock(name="generic_part")** indicates PROMNeT++ that the model's generic part ends at this point. It must be used in conjunction with @BeginTemplateBlock.

[2]A good example of this is JPA's *@Table* annotation. See: `http://www.objectdb.com/api/java/jpa/Table`.

[3]`https://code.google.com/p/promnetpp/wiki/PROMNeTppAnnotations`

Chapter 5

Risk management

5.1 Defining "(thresold of) success" and "risk"

The notion of "risk", as known by modern software engineers, depends on the software project at hand. To define "risk" for this project, one must define its threshold of success first. The threshold of success for this is defined below.

"Success", in this project's context, is achieved if, and only if, all of the following conditions are met:

1. The translation process, from PROMELA code to C++ code, works for every round-based consensus protocol that is to be studied. Every protocol is regarded as a test case for the translator tool, and the tool must produce C++ code from the protocol's PROMELA specification without fail.[1]
2. The resulting C++ code and miscellaneous OMNeT++ specific files (such as OMNeT++'s network layout file) must allow the user to perform one or multiple error-free simulations in OMNeT++. That is, the user, after waiting for the tool to finish the source code generation process, must have all the necessary files to run a simulation in OMNeT++.

Given the above threshold of success, "risk" is, thus, defined as any foreseeable problem that might impede success.

5.2 Risk list

The following tables describe several risks associated with either the project's nature or its development phase. The template used here is adapted from OpenUP's[2].

Some of the risks listed here have been preemptively mitigated during the first semester of the internship, so as to establish a suitable course of action towards achieving the aforementioned threshold of success. Also, due to technical restrictions, only 4 columns at a time can be visible;

[1]Do note, however, that the tool may still fail *before* the actual translation phase, for instance, if Spin reports that there is not enough memory to verify the PROMELA model.

[2]`http://epf.eclipse.org/wikis/openup/core.mgmt.common.extend_supp/guidances/templates/risk_list_33A6AE1E.html`

as such, it was necessary to split the original template's table into multiple tables.

Risk ID	**Date identified (DD-MM-YYYY)**	**Headline**	**Description**
1	27-01-2013	Development is not proceeding as scheduled	Only one person (Miguel Martins) would be actively working on this project, and as such, there is always the possibility that he won't work at an adequate pace, increasing the risk of finishing the second semester with an incomplete tool.
2	22-01-2013	Absence of a suitable network simulator	A network simulator that is capable of simulating protocols at an applicational level is either unknown or does not exist at all.
3	27-01-2013	JavaCC proves to be difficult to use	The author had little to no experience with JavaCC prior to this project, and thus JavaCC's learning curve was unknown to him.
4	27-01-2013	Artifacts and/or source code files for the project are lost	As with any software project, there is always the concern about losing files; the SVN repository provided by Helpdesk may be inaccessible for the purpose of committing and/or retreiving files. Software/hardware failures on the developer's side may also result in file loss.
5	27-01-2013	Implementing/translating a specific Spin feature/behavior proves to be either difficult or very complex	Spin/PROMELA has certain features or behaviors whose mapping to C++ code is not direct/trivial. The most notable of these is the fact that there is non-determinism in Spin, such as do/if statements with multiple guards. In some cases, there is one particular guard that overlaps with another, and in other cases no guard is executable, in which case the process should block until at least one guard becomes executable.
6	22-01-2013	Learning how to effectively use Spin proves to be a difficult task	Spin, like most software tools, has a learning curve associated with it; knowing how to work with Spin is crucial to determine how PROMELA code should be translated.

Risk ID	Type	Impact (1-5)	Probability
1	Organizational	5	0.2
2	Technical	5	0.05
3	Technical	5	0.04
4	Technical	5	0.03
5	Technical	4	0.1
6	Organizational	2	0.01

5.2.1 Risk response strategies

Risk ID	Strategy	Strategy's description
1	Mitigate	Commit artifacts/code frequently to the SVN repository, at least once a day. Revise the personal work schedule, and allocate more hours of work per day as deemed necessary.
2	Mitigate	Analyze network simulation alternatives via web; this has already been done, as it has been determined that OMNeT++ seems to be the most suitable alternative.
3	Mitigate	Search and read tutorials for JavaCC; various tutorials are readily available online, such as Theodore Norvell's. Several JavaCC-related books are also available, most notably Tom Copeland's "Generating Parsers with JavaCC, Second Edition"; the Department of Informatics Engineering's library also has books on JavaCC, readily available. Also, JavaCC's mailing lists can be used as necessary.
4	Mitigate	Use suitable backup schemes to minimize the chances of file loss. One possible way of doing this is to have the directory for an SVN repository in the cloud so that, should the SVN repository be inaccessible, files can still be read/written to without ceasing development. Services like Dropbox (which the author frequently uses, and has used for this project as well) are often organized so that files are distributed across multiple geographical locations. In addition, a manual backup policy could be applied to further minimize file loss: for instance, archive and compress the contents of the entire SVN repository to a 7-zip file, and copy it to a USB flash drive, or even to a second cloud storage service such as Google Drive.
5	Mitigate	Discuss implementation issues with the supervisor, Raul Barbosa, who has extensive knowledge on Spin. Alternatively, post questions on Spinroot.com's forums.
6	Mitigate	Study Spin as intensively as needed; this has been done, in part, as the author started to familiarize himself with Spin towards the end of September 2012. Additionally, the author has also attended Raul Barbosa's Model Checking classes at the Department of Informatics Engineering, which are part of the "Dependable Computer Systems" PhD. subject, during the first semester.

5.3 Reflecting over the risks

None of the six risks presented in the previous section have truly impeded "success", as defined in section 5.1. However, risk number 5 has manifested itself on occasion, in the form of technical implementation details. At one point in development, the *receive* routine was being treated as a blocking routine, requiring each process to save its own progress in the *state_transition* routine (which calls *receive* internally). Moreover, *receive* was being called within a loop, making the following code unsuitable:

```
void Process::state_transition() {
(...)

for (i = 0; i <= (NUMBER_OF_PROCESSES - 1); ++i) {
    if (my_state.received_message[i]) {
        receive(_message, i); /* Suspends the execution of this
        process, making it so that, when re-entering this routine,
        it must enter the for loop again. */

        my_state.values[i] = _message.value;
        my_state.received_message_count++;
    }
}

(...)
```

Initially, this was solved by using a mechanism known as a *step map*, which stores every step (an integer value) for every routine, for all processes.

```
void Process::state_transition() {
(...)

int step = step_map["state_transition"];
if (step == 0) {
    i = 0;
    ++step_map["state_transition"];
    ++step;
}

if (step == 1) {
    if (my_state.received_message[i]) {
        receive(_message, i); /* May block and suspend the
        routine */
        step_map["state_transition"] = 2;
    } else {
        step_map["state_transition"] = 3;
    }
}

if (step == 2) {
    my_state.values[i] = _message.value;
    my_state.received_message_count++;
    if (i <= (NUMBER_OF_PROCESSES - 1)) {
        ++step;
    }
}

if (step == 3) {
    ++i;
    step_map["state_transition"] = 1;
}

(...)
```

The above code essentially breaks down the “for” loop into a sequence of steps, so that even if *receive* is called, *state_transition* can be re-entered from the point where it left off. Naturally, this would not only make the translated code less appealing to read, it would also introduce complexity in the routine. This problem was eventually solved by turning *receive* into a non-blocking routine, and storing incoming messages via a special, separate routine named *enqueue_message*.

Chapter 6

Work and results

Using what's been described in this report so far, PROMNeT++ was incrementally developed until it reached its current version to date: PROMNeT++ Beta 6. This version is, of course, downloadable at its respective Google Code page[1], packaged as a ZIP file. It is distributed with two distinct round-based consensus protocols.

One of the round-based consensus protocols that PROMNeT++ is distributed with goes by the name of *OneThirdRule*. This protocol was implemented, in PROMELA, by Raul Barbosa, and later adapted for use with PROMNeT++ by Miguel Martins; the PROMELA implementation for this protocol is contained within the *NewOneThirdRule.pml* file. This protocol originates from Charron-Bost and Schiper's article on "The Heard-Of model", and is described, in their terms, as "a very simple algorithm that does not require any coordinator election procedure"[36].

The other round-based consensus protocol found inside the ZIP file is named *1-of-N*, and was also adapted for use with PROMNeT++ from Raul Barbosa's original implementation of it, much like the *OneThirdRule* protocol; the file *1-of-n.pml* contains its implementation in PROMELA. It originates from Negin Fathollahnejad et al.'s work on a "probabilistic Analysis of a 1-of-N selection algorithm".[37]

Thus, we had two distinct PROMELA models (one per protocol) to work with. Our main goal was to ensure that both of them were translated to valid C++ implementations. As mentioned in chapter 3, we can compare the output from PROMELA/Spin to the output from C++/OMNeT++, to assert the validity of the translation process; if, for 30 different seeds, there is always a match between outputs, then we can confirm that the generated C++ code is an implementation of the PROMELA model.

6.1 McCabe complexity for the generated source code

As expected, PROMNeT++ does not add a significant amount of complexity to the original PROMELA models; on the contrary, any complexity, if it exists, is minuscule. This is evidenced by the cyclomatic complexity values presented in the table below. These values were obtained

[1]`https://code.google.com/p/promnetpp/downloads/detail?name=promnetpp-beta-6.zip`

with the aid of Metriculator[2], an Eclipse plug-in that gathers metrics for C++ source code. Fortunately, the OMNeT++ IDE is Eclipse-based, and this plug-in can be installed and run inside the IDE without errors.

Protocol	Routine	McCabe value
OneThirdRule	compute_message	1
OneThirdRule	state_transition	11
OneThirdRule	system_every_round	6
OneThirdRule	system_init	3
1-of-N	compute_message	2
1-of-N	state_transition	21
1-of-N	system_every_round	6
1-of-N	system_init	3

Do note that any other routines (such as *begin_round*, and *end_round*), although part of the PROMELA structure for round-based consensus protocols, are included in the generic part of the models, and are thus not dynamically translated to C++ code. We can clearly see, in the table above, that McCabe's complexity seldom exceeds the value of 10 units. It is known that McCabe himself had proposed the value of 10 as a "reasonable (...) upper limit"[38]; even OneThirdRule's *state_transition* routine does not greatly exceed said limit (only by 1 unit).

The only "critical" instance here is 1-of-N's *state_transition* routine, with a McCabe value of 21. However, by simply observing the original code for 1-of-N's *state_transition*, we can clearly see that most of the complexity comes from the original routine, and not the translation process. Consider, for instance, the following "if" block in 1-of-N's PROMELA specification, part of its *state_transition* inline:

```
if
    :: round < (R-1) ->
        round++;
        for(i : 0..(NUMBER_OF_PROCESSES-1)) {
            if
            :: my_state.received_message[i] ->
                receive(_message, i);
                if
                :: my_state.local_value < _message.value ->
                    my_state.local_value = _message.value
                :: else -> skip
                fi;

                for(j : 0..(NUMBER_OF_PROCESSES-1)) {
                    my_state.view[j] = my_state.view[j] ||
                        _message.view[j]
                }
                :: else -> skip
            fi
        }
(...)
fi;
```

[2]http://marketplace.eclipse.org/content/metriculator

We can clearly see that there's a "for" loop, within an "if" block, which in turn is contained within an outer "for" loop, which in turn is contained within the outermost "if" block. And thus, it becomes clear that 1-of-N's *state_transition* is, on its own, a complex routine.

6.2 Verification by output comparison

Earlier in this document, in chapter 3, we have described an experiment that consists on choosing a seed for the custom pseudo-RNG contained within the PROMELA model, then attempting to match Spin's output with that of OMNeT++. An example was shown, using the value 1234 as the LCG's seed, for the OneThirdRule protocol. Both outputs match, and thus we consider that the translation process for the OneThirdRule protocol, with a seed of 1234, produces a valid implementation of the model.

From a statistical point of view, however, one experiment alone (i.e. one seed, one protocol) should not be regarded as sufficient to validate PROMNeT++'s translation process. Given that we have another protocol at our disposal, and that we can change the value of the seed as necessary, the very same experiment was performed on 30 distinct seeds, for both the *OneThirdRule* and the *1-of-N* protocols. The values for the seeds are as follows:

```
1234, 71337, 749464, -252392, -355723, 960103, 905902, 634195, -807626, 458852,
-438956, 521259, -231442, 615387, 392039, -456988, 144748, 685910, 115335,
-481879, -145600, -20244, 569789, 980987, 916986, 560451, 868386, 568700,
-165345, -47588
```

In total, 120 text files[3] have been produced, both via Spin and OMNeT++. They are not shown in this document, and are instead distributed as attachments, in CD-ROM format. It is entirely possible to produce them on one's own, however, as the following subsection describes.

6.2.1 Producing output files

Using Spin

Producing output with Spin is trivial for both the *OneThirdRule* and *1-of-N* protocols. Using the file *1-of-n.pml* as an example, we may open a terminal (such as Windows 7's command prompt), navigate to where it is located, and issuing the following command:

```
spin 1-of-n.pml > output.txt
```

Then, we edit the *1-of-n.pml* with a text editor (such as Notepad++), and replace the current seed with a new one. For example, if the initial seed was 1234, and we wish to test the seed 71337 next, we replace the following line

```
int rnd = 1234;
```

with

```
int rnd = 71337;
```

[3]There are 60 output files per protocol; there's 30 seeds, 2 distinct applications for producing the output (Spin and OMNeT++), and 2 distinct protocols, making a total of $30 \times 2 \times 2 = 120$ text files in total.

We may then repeat the first command (using a different file name for the output), and continue doing this process until all seed values are exhausted.

For the *OneThirdRule* protocol, the first command needs a small tweak; since the protocol never terminates (unlike 1-of-N, which terminates due to "timeout", by design), we must limit Spin's number of steps. This is done by passing *-uN* as a command-line argument for Spin, where *N* is the number of steps, and can be integer value greater than zero. During our experiments, we observed that 10000 (ten thousand) steps are more than enough for this purpose, as it allows all processes to reach consensus, regardless of what seed is used. Thus, for the OneThirdRule protocol, output is produced as follows:

```
spin -u10000 NewOneThirdRule.pml > output.txt
```

Using PROMNeT++ and OMNeT++

The procedure here is nearly as straight-forward as it is when using Spin, since PROMNeT++ is entirely command-line based. The first step is to translate one of the PROMELA models to C++ code. Using the *NewOneThirdRule.pml* file as an example:

```
set PROMNETPP_HOME=C:\promnetpp
java -enableassertions -jar "%PROMNETPP_HOME%\promnetpp.jar" NewOneThirdRule.pml
```

Or, for Linux/OS X users:

```
export PROMNETPP_HOME=/Users/yourusername/promnetpp
java -enableassertions -jar $PROMNETPP_HOME/promnetpp.jar NewOneThirdRule.pml
```

Then, assuming PROMNeT++ did not report any errors, the next step is to copy the generated files to an existing OMNeT++ project (which can be created from within the IDE, by selecting File—New—OMNeT++ Project...). This can usually be achieved by manually dragging the files over to OMNeT++'s IDE, and dropping them on the project's folder.

The final step is to build the OMNeT++ project (Right Click—Build Project), and performing a simulation (Right Click—Run As—OMNeT++ Simulation). Unfortunately, no automated process was discovered for producing output here, so simulations in OMNeT++ are manually started and stopped, after a few rounds (10 rounds are enough for any of the two protocols to reach consensus). A file with the name *simulation-output.txt* is created and written to during the simulation process, containing the output from the translated *printf* statements.

This procedure is described in more detail at PROMNeT++'s Wiki, in a page labelled "CommandLineWorkFlow1"[4].

6.2.2 Comparing the produced output files

Using the procedure described in the previous section, we can produce two distinct output files: one related to PROMELA/Spin, and another related to PROMNeT++/OMNeT++. We need to compare them to each other, meaning that we must determine if their contents are equal. Note, however, that "equality" here does not account for the order of which some of the lines

[4] https://code.google.com/p/promnetpp/wiki/CommandLineWorkflow1

appear; as mentioned in chapter 3, processes may execute in a certain order when simulating via Spin, while executing in a different order in OMNeT++. This makes it so that, when looking at both outputs side by side, it is likely that some of the text lines appear to be "swapped".

As such, we define that both files are equal if their *sorted* contents are equal. By submitting both files to a command-line tool for sorting lines of text, such as GNU's sort or Windows' sort command, and then computing the difference between sorted outputs, via GNU's *diff*, one can effectively determine is there is a match between both files; if *diff* does not produce output, both files are equal; otherwise, they are not. Miguel Martins has written a batch script for Windows for this very purpose, aptly named *sort and compare.bat*. The batch code for it is as follows:

```
@echo off
set GNUWIN32_HOME="C:\Program Files (x86)\GnuWin32"
set PATH=%PATH%;%GNUWIN32_HOME%\bin
set SORT_AND_COMPARE_HOME="C:\sort and compare"
REM ----------------------------------------------------------------------------
sort %1 > %SORT_AND_COMPARE_HOME%\temp1.txt
sort %2 > %SORT_AND_COMPARE_HOME%\temp2.txt
cd %SORT_AND_COMPARE_HOME%
diff temp1.txt temp2.txt > diff.txt
del temp1.txt
del temp2.txt
for %%A in (diff.txt) do set SIZE=%%~zA
del diff.txt
if %SIZE% EQU 0 (
    echo Files are equal.
) else echo Files are not equal.
pause
```

The above script uses Windows 7's built-in *sort* command, as well as GnuWin32's *diff*[5], the latter being a port of GNU's *diff* to the Windows family of operating systems. Using the above script, the procedure becomes clear: one has to merely submit two text files as arguments, one resulting from Spin, one resulting from OMNeT++; the script automates the work for the user, and either outputs one of "Files are equal" or "Files are not equal". Fortunately for us, when experimenting with the 30 seed values above, for both round-based consensus protocols, the above script has always reported equality, thus validating PROMNeT++'s translation process.

[5]http://gnuwin32.sourceforge.net/packages/diffutils.htm

Chapter 7

Limitations and future work

It is of the most importance to note that PROMNeT++ should not (and is not) regarded as a flawless PROMELA to C++ translation tool. It is, in fact, associated with a set of limitations described in this chapter.

First and foremost, PROMNeT++ is restricted to translating PROMELA models pertaining to round-based consensus protocols, rather than every single possible PROMELA input, meaning that any user is limited to said type of models when using it. As shown in the previous chapter, PROMELA models for PROMNeT++ have a very specific, well-defined structure that *must* be followed; PROMNeT++ will not attempt to perform an accurate PROMELA to C++ translation for any model that does not comply with said structure.

Furthermore, PROMNeT++ is built with a parser for PROMELA that was *written from scratch* during the second semester of this work's internship, using JJTree. This parser does not implement the entirety of the PROMELA grammar[1], leaving out, for instance, keywords such as *xs* and *xr* (exclusive send and exclusive receive, respectively). In fact, this JJTree-based parser was incrementally coded with only the sufficient grammatical elements so that our round-based consensus protocol PROMELA models would be parsed without errors. Thus, said parser can be seen as a "best-effort" solution for the purposes of this project.

Likewise, PROMNeT++ itself was written from scratch, and is the result of the development efforts of one single person (Miguel Martins). Ultimately, PROMNeT++ is not guaranteed to accurately map every single PROMELA feature or behavior to C++ code. Perhaps the most notable example of this is the absence of blocking primitives/mechanisms that exist in PROMELA, but do not exist in PROMNeT++. Consider the following:

```
/* This is PROMELA code */
proctype MyProcess() {
    (...)

    if
        :: condition1 -> printf("Action 1.\n");
        :: condition2 -> printf("Action 2.\n");
        :: condition3 -> printf("Action 3.\n");
    fi
```

[1] http://spinroot.com/spin/Man/grammar.html

```
}
```

Unlike what happens in either C or C++, when all of the above conditions are evaluated as "false", *MyProcess* becomes blocked within the "if" block until any of the conditions becomes "true". PROMNeT++ would translate the above "if" block like so:

```
/* This is the C++ code that PROMNeT++ would generate */
if (condition1) {
    utilities::printf("Action 1.\n");
} else if (condition2) {
    utilities::printf("Action 2.\n");
} else if (condition3) {
    utilities::printf("Action 3.\n");
}
```

Note that, in the above generated code, nothing happens when all three conditions are "false". A hypothetical solution to this problem would be:

```
while (true) {
    if (condition1) {
        utilities::printf("Action 1.\n");
        break;
    } else if (condition2) {
        utilities::printf("Action 2.\n");
        break;
    } else if (condition3) {
        utilities::printf("Action 3.\n");
        break;
    } else {
        block(); /* Block until any of the above conditions
        becomes true */
    }
}
```

While it may look like an appealing solution, implementing a *block()* routine during development ultimately proved to be impossible, if not extremely complex. Languages such as C, or in this case C++, do not possess any blocking mechanisms, unless executing within a multi-threaded environment, such as the well-known *pthread_cond_wait()* routine from POSIX Threads. For this project, this option was entirely unavailable, due to the fact that OMNeT++, much like the vast majority of simulation environments, is single-threaded.

Thus, as future work, we hereby propose that, whenever possible:

- PROMNeT++ be extended to more kinds of communication protocols.
- PROMNeT++'s JJTree-based parser be augmented to accommodate the full PROMELA grammar.
- PROMNeT++ be extended (or possibly re-written) to target a pure multi-threaded environment, eliminating the need for OMNeT++ entirely; for example, using POSIX Threads, each node in the communication protocol could be mapped to a thread; threads

would communicate either via pipes or shared memory, with the appropriate synchronization mechanisms.

We hereby remind the reader here that, as mentioned in chapter 1 of this document, all of the source code for PROMNeT++ may be obtained, modified and re-used completely free of charge, under the MIT License, and encourage any developers interested in the topic to come up with their own solutions to the above limitations.

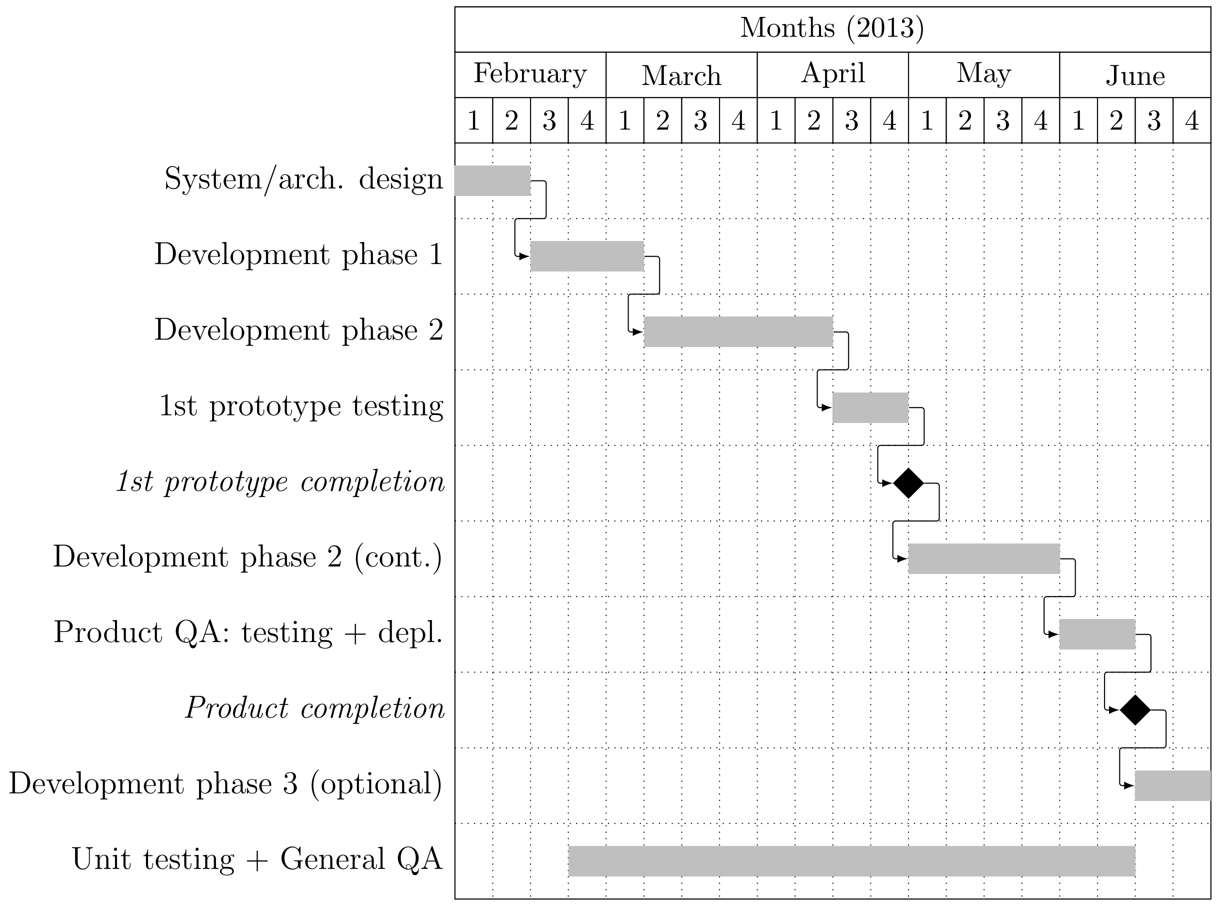
Chapter 8

Planned work versus actual work

During the first semester of the internship, a work plan had been formed, as the Gantt chart below illustrated. Product development was divided into 3 phases. Phase 1 would consist in writing code templates for OMNeT++: pre-made C++ code containing the basic structure and functionality for round-based consensus protocols. Phase 2 would deal with the actual coding of PROMNeT+, using the templates produced in phase 1. Finally, phase 3 would consist of the integration of the translator tool with the Eclipse IDE, and was marked as optional, as integration with Eclipse is not mandatory for this project, but is rather a "bonus feature"; therefore, if needed, phase 2 could have been extended 2 weeks further and phase 3 may not have taken place at all.

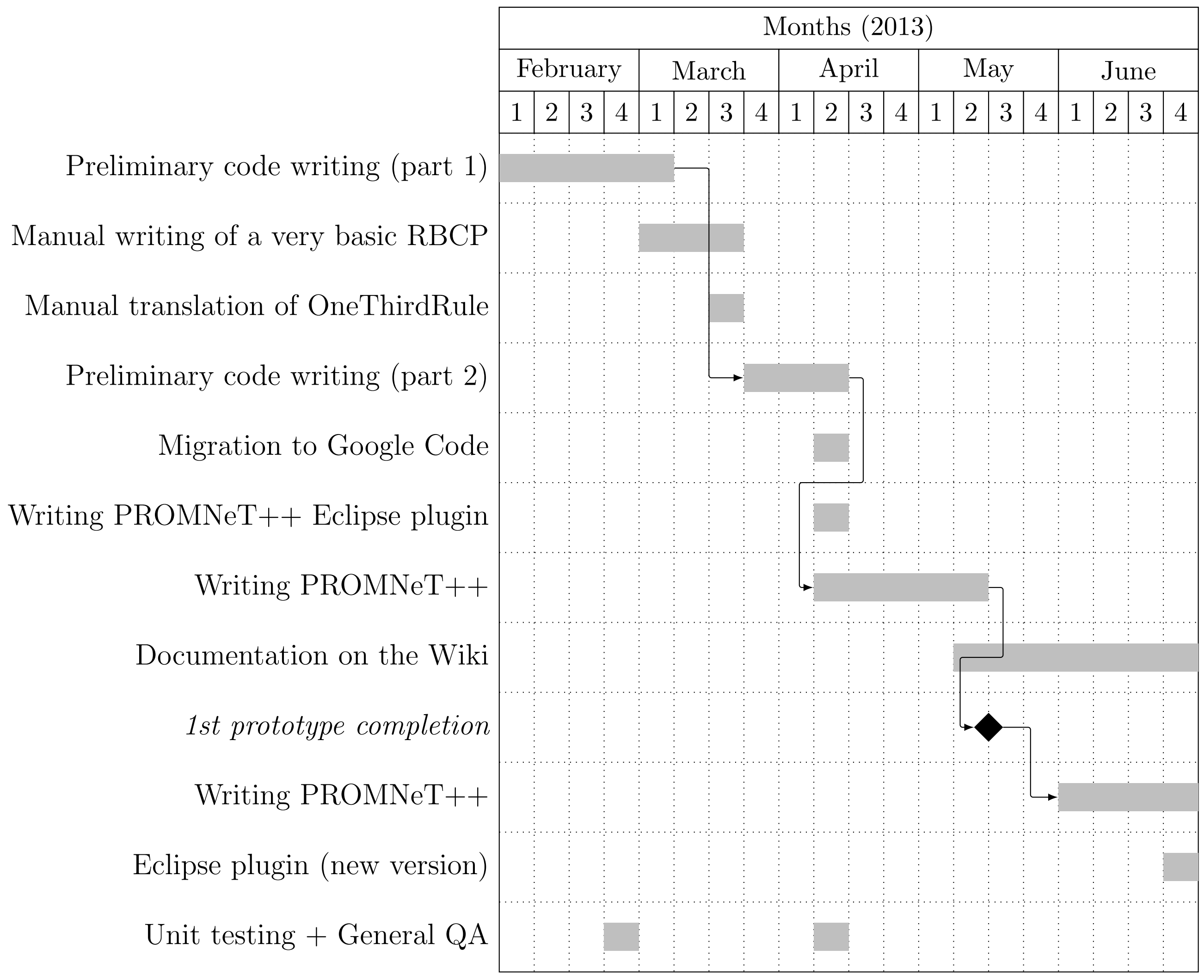
Due to technical restrictions, some abbreviations for the below Gantt charts' labels are used, namely "RBCP" for "Round-Based Consensus Protocol", "arch." for "architecture", and "depl." for "deployment".

8.1 Planned work

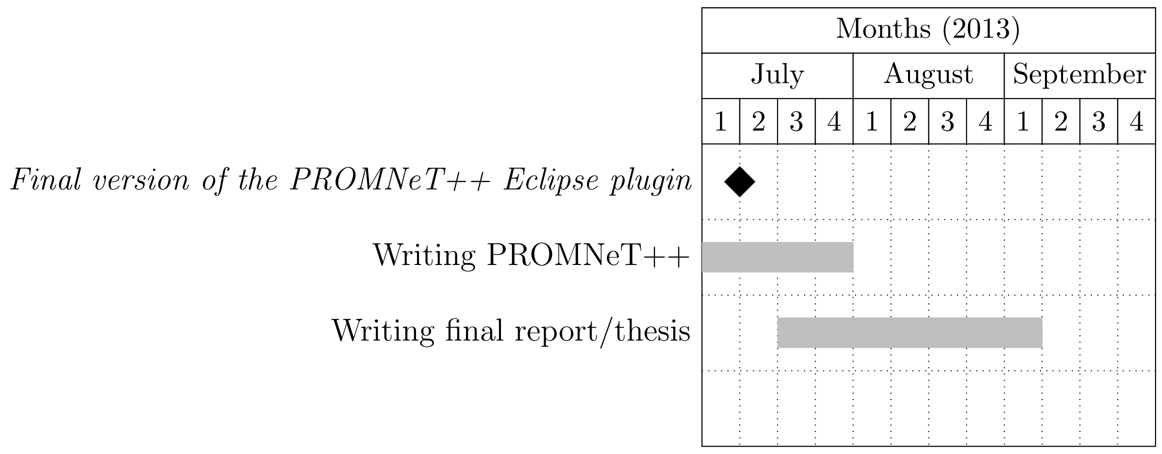


Development for PROMNeT++ started during the second semester, on February 12, 2013. Preliminary code for PROMNeT++ was hosted on the SVN repository provided by the Department of Informatics Engineering, before migrating to its current repository provided by Google Code. The Gantt charts below are a representation of Miguel Martins' actual effort.

8.2 Actual work (February 2013 to June 2013)



8.3 Actual work (July 2013 to September 2013)



Chapter 9

Conclusions

Throughout this document, we have presented PROMNeT++, an open-source tool developed by Miguel Martins, to ascertain whether or not it is possible to generate runnable source code from round-based consensus protocols modelled in PROMELA. For the *OneThirdRule* and *1-of-N* protocols distributed with PROMNeT++'s latest version to date [1], this is certainly possible. Since these are two distinct protocols which make use of the typical constructs available in the PROMELA language, we believe that many other round-based protocols can be translated using this tool.

All the essential requirements (i.e. those that have not been labelled as "optional" in chapter 4) have been met. A functional Eclipse plug-in that uses PROMNeT++, which was one of the non-essential requirements (a "bonus feature"), is also available at PROMNeT++'s Download page. We believe we are now in possession of a tool that will promote future research on the topic of the automatic generation of source code from formal models. In fact, we encourage future internship students and/or outside developers to modify and/or extend PROMNeT++ as necessary.

As a small reminder on the importance of this project's research topic, it is important to mention here that said automatic generation of source code is a powerful technique to deliver software with very high quality standards, overcoming the defects of more common techniques such as unit testing. In a way, PROMNeT++ is a step towards encouraging software developers to use this technique, and produce less faulty software.

An extensive effort was made to guarantee the quality of the code generated by the tool. We have, at our disposal, the simulation output of the model and the respective code we compared in numerous test scenarios, so as to ensure that the generated code is an implementation of the model. Furthermore, the generated code was reviewed and analysed in order to ensure that its complexity remains similar to the original models. This gives us some confidence that the generated code can be useful not only for performing simulation with OMNeT++ but also for execution in real systems.

[1] `https://code.google.com/p/promnetpp/downloads/detail?name=promnetpp-beta-6.zip`

Bibliography

[1] Edmund M. Clarke, Orna Grumberg, Doron A. Peled. *Model Checking*, MIT Press (2000)

[2] A. Avizienis, J. C. Laprie, B. Randell, C. Landwehr *Basic Concepts and Taxonomy of Dependable and Secure Computing*, IEEE Transactions on Dependable and Secure Computing (2004), pages 12-13

[3] Dahl O.J, Dijskstra E.W., Hoare C.A.R. *Structured Programming*, Academic Press (1972)

[4] Edmund M. Clarke *Assuring Software Quality by Model Checking*, Carnegie Mellon University (2002) (presentation URL: `http://laser.inf.ethz.ch/2011/Elba/clarke/New%20Lecture%201.pdf`)

[5] R. Barbosa, J. Karlsson *Formal Specification and Verification of a Protocol for Consistent Diagnosis in Real-Time Embedded Systems*, Chalmers University of Technology (2008), pages 222–223

[6] E. Gafni *Round-by-Round Fault Detectors: Unifying Synchrony and Asynchrony*, ACM (1998)

[7] W. Visser, M. B. Dwyer, M. Whalen *The hidden models of model checking*, Springer (2012), page 2

[8] G. J. Holzmann *The Model Checker SPIN*, IEEE Transactions on Software Engineering

[9] K. L. McMillan *The SMV system for SMV version 2.5.4*, Carnegie Mellon University (2000)

[10] A. Sinha, A. Ry, S. Singh *Modeling & Verification of Sliding Window Protocol with Data Loss and Intruder Detection using NuSMV*, Manipal Institute of Technology, Manipal University (2012)

[11] V. Vishal, S. Gugwad, S. Singh *Modeling and Verification of Agent based Adaptive Traffic Signal using Symbolic Model Verifier*, arXiv (2012)

[12] G. Behrmann, A. David, K. G. Larsen *A Tutorial on Uppaal 4.0*, Department of Computer Science, Aalborg University (2006)

[13] O. Wibling, J. Parrow, A. Pears *Automatized Verification of Ad Hoc Routing Protocols*, Department of Information Technology, Uppsala University (2004)

[14] A. Nimiya, T. Yokogawa, H. Miyazaki, S. Amasaki, Y. Sato, M. Hayase *Model checking consistency of UML diagrams using Alloy*, World Academy of Science, Engineering and Technology (2010)

[15] S. Pai, Y. Sharma, S. Kumar, R. M Pai, S. Singh *Formal Verification of OAuth 2.0 using Alloy Framework*, 2011 International Conference on Communication Systems and Network Technologies (2001)

[16] I. Feinerer, G. Salzer *A comparison of tools for teaching formal software verification*, Formal aspects of computing (2009)

[17] M. Sample, G. Neufeld *Snacc 1.1: A high performance ASN. 1 to C/C++ compiler*, Vancouver: University of British Columbia (1993)

[18] J. Levine *flex & bison*, O'Reilly Media (2009), pages 81–117

[19] E. Mamas, K. Kontogiannis *Towards Portable Source Code Representations Using XML*, University of Waterloo (2000), pages 6–7

[20] E. M. Gagnon, L. J. Hendren *SableCC, an Object-Oriented Compiler Framework*, McGill University (1998)

[21] N. Li, M. Shen, S. Li, L. Zhang, Z. Li *STVsm: Similar Structural Code Detection Based on AST and VSM*, Springer (2012)

[22] C. Wulf, S. Frey, W. Hasselbring *A Three-Phase Approach to Efficiently Transform C# into KDM*, University of Kiel (2012)

[23] R. Martín Brualla *Automatic translation of programs for evaluation of execution times*, Universitat Politècnica de Catalunya (2011)

[24] R. Herzog, K. Schuhmann, D. Schwudke, J. L. Sampaio, S. R. Bornstein, M. Schroeder, A. Shevchenko *LipidXplorer: a software for consensual cross-platform lipidomics*, PLoS One (2012)

[25] A. M. Kanthe, D. Simunic, R. Prasad *A Mechanism for Gray Hole Attack Detection in Mobile Ad-hoc Networks*, International Journal of Computer Applications (2012)

[26] D. Sivaganesan, R. Venkatesan *Dynamic Cluster Routing Protocol for Broadcasting in Clustered Mobile Ad Hoc Networks*, European Journal of Scientific Research (2012)

[27] A. Derhab, F. Ounini, B. Remli *MOB-TOSSIM: An Extension Framework for TOSSIM Simulator to Support Mobility in Wireless Sensor and Actuator Networks*, IEEE 8th International Conference (2012)

[28] E. Perla, A. Ó Catháin, R. S. Carbajo, M. Huggard, C. Mc Goldrick *PowerTOSSIM z: realistic energy modelling for wireless sensor network environments*, ACM (2008)

[29] O. Landsiedel, H. Alizai, K. Wehrle *When Timing Matters: Enabling Time Accurate and Scalable Simulation of Sensor Network Applications*, IPSN'08. International Conference on IEEE (2008)

[30] M. Dräxler, F. Beister, S. Kruska, J. Aelken, H. Karl *Using OMNeT++ for Energy Optimization Simulations in Mobile Core Networks*, ACM (2012)

[31] A. Comsa, A. B. Rus, V. Dobrota *Simulation of the Floyd-Warshall Algorithm Using OMNeT++ 4.1*, 9th International Conference on Communications (COMM) (2012)

[32] A. Iliasov *Generation of certifiably correct programs from formal models*, Software Certification (WoSoCER), 2011 First International Workshop on. IEEE (2011)

[33] L. Lensink, C. Munoz, A. Goodloe *From Verified Models to Verifiable Code*, NASA, Langley Research Center, Hampton VA 23681-2199, USA (2009)

[34] T. Tsuchiya, A. Schiper *Using Bounded Model Checking to Verify Consensus Algorithms*, Springer (2008)

[35] O. Mürk, D. Larsson, R. Hähnle *KeY-C: A Tool for Verification of C Programs*, Automated Deduction–CADE-21 (2007)

[36] B. Charron-Bost, A. Schiper *The Heard-Of model: computing in distributed systems with benign faults*, Springer (2009), page 69

[37] N. Fathollahnejad, E. Villani, R. Pathan, R. Barbosa, J. Karlsson *Probabilistic Analysis of a 1-of-N Selection Algorithm Using a Moderately Pessimistic Decision Criterion*, 19th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2013), to appear

[38] T. McCabe *A Complexity Measure*, IEEE Transactions on Software Engineering (1976)

Appendix A

User feedback on PROMNeT++'s translation process

Raul Barbosa, this internship's supervisor, had arranged an informal meeting with researchers Fatemeh Ayatolahi and Behrooz Sangchoolie, to determine how feasible it would be to re-use PROMNeT++'s generated source code, and ultimately run a round-based consensus protocol on their system, without resorting to OMNeT++.

It is important to note, first and foremost, that their system is setup to compile and run C code, rather than C++ code (which OMNeT++ uses). Thankfully, most of the generated source code, and in particular the source code for the *compute_message*, *state_transition*, *system_every_round* and *system_init* routines, is backwards compatible with C, with minor adaptations (such as using "printf()" instead of "utilities::printf()").

Raul Barbosa was also inquired as to how we guarantee that the generated code is a valid implementation of a PROMELA model, which was adequately answered by referring to the output comparison experiment in this document. Lastly, it was noted that, in order to make use of the generated source code, a runtime environment for their system would have to be built from scratch. Said runtime environment would be responsible for handling low-level events, such as sending and receiving messages, as well as ensuring that each node in the system is correctly initialized and runs the appropriate routines in the correct order. This was to be expected, due to the fact that OMNeT++ provides its users with a high-level API for message handling and node initialization, which would normally not exist in a real-time system.

Fortunately, it was noted that doing the above is "not very complex", and as such, all that's truly required is that the generated code for the various round-based consensus protocol routines functions correctly. Additionally, they have mentioned that they would like to map one process to one node, while all remaining processes are mapped to another node, allowing them to monitor the state of the single-process node. A simple solution to this problem would be to insert multiple instances of the same code in each node.

Overall, PROMNeT++ seems to generate source code without requiring a considerable amount of effort to be re-used in "OMNeT++-less" environments.

Appendix B

The OneThirdRule protocol, translated

To better understand how PROMNeT++ operates, this appendix shall illustrate how the *OneThirdRule* protocol is translated, from PROMELA to C++. The starting PROMELA code is as follows:

```
/*
 * Copyright (c) 2013, Raul Barbosa
 * Use is subject to license terms.
 *
 * This source code file is provided under the MIT License. Full
 * licensing terms should be available in the form of text files.
 * The standard source code distribution provides a LICENSE.txt
 * file which can be consulted for licensing details.
 */

/*
OneThirdRule in PROMELA
Author: Raul B. Barbosa <rbarbosa@dei.uc.pt>
Modified by Miguel Martins <marm@student.dei.uc.pt>

Modifications were done on the specification provided on April 23
2013:
-More descriptive variable names
-Indentation of 4 spaces per tab
-Annotations for PROMNeT++
*/

/* @UsesTemplate(name="round_based_protocol_generic") */
/* @TemplateParameter(name="numberOfParticipants") */
#define NUMBER_OF_PROCESSES 3
#define NUMBER_OF_ASYNCHRONOUS_ROUNDS 1

/* Random number generation */
int rnd = 1234;
#define next(r) (r * 16807) % 2147483647
#define boolean(r) ((r >> 30) & 1)

/* End random number generation */
```

```
int round_id = 0;

typedef message {
    byte value;
}

typedef process_state {
    bool received_message[NUMBER_OF_PROCESSES];
    byte received_message_count;

    byte local_value;
    byte decision_value;
    byte values[NUMBER_OF_PROCESSES]
}

process_state state[NUMBER_OF_PROCESSES];

#define my_state state[_pid-1]

inline compute_message(_message) {
    _message.value = my_state.local_value
}

inline state_transition() {
    d_step {
        my_state.received_message_count = 0;
        for(i : 0..(NUMBER_OF_PROCESSES-1)) {
            if
            :: my_state.received_message[i] ->
                receive(_message, i);
                my_state.values[i] = _message.value;
                my_state.received_message_count++
            :: else -> skip
            fi
        }

        if
        :: my_state.received_message_count > (2 *
            NUMBER_OF_PROCESSES/3) ->
            l = 0;
            for(i : 1..(NUMBER_OF_PROCESSES)) {
                k = 0;
                for(j : 0..(NUMBER_OF_PROCESSES-1)) {
                    if
                    :: my_state.values[j] == i -> k++
                    :: else -> skip
                    fi
                }
                if
                :: k > l -> my_state.local_value = i; l = k
                :: else -> skip
                fi
            }
        :: else -> skip
        fi;

        if
        :: l > (2 * NUMBER_OF_PROCESSES/3) ->
            assert((my_state.decision_value ==
                my_state.local_value)
                || (my_state.decision_value == 0));
            my_state.decision_value = my_state.local_value;
            printf("MSC: P%d decides %d on round %d\n", _pid,
                my_state.decision_value, round_id)
        :: else -> skip
        fi
    }
}

inline system_init() {
    j = 1;
    for(i : 0..(NUMBER_OF_PROCESSES-1)) {
        state[i].local_value = j;
        rnd = next(rnd);
        if
        :: boolean(rnd) -> j++
        :: else -> skip
        fi;
        printf("MSC: P%d has initial value x=%d\n", i+1,
            state[i].local_value)
    }
}

inline system_every_round() {
    round_id++;
    printf("MSC: new round, id=%d\n", round_id);
    printf("rnd=%d\n", rnd);

    if
    :: remaining_asynchronous_rounds == 0 -> synchronous = true
    :: else -> remaining_asynchronous_rounds--
    fi;

    for(i : 0..(NUMBER_OF_PROCESSES-1)) {
        for(j : 0..(NUMBER_OF_PROCESSES-1)) {
            if
            :: synchronous || i == j ->
                state[i].received_message[j] = true
            :: else ->
```

```
                rnd = next(rnd);
                state[i].received_message[j] = boolean(rnd)
            fi
        }
    }
}

/* @BeginTemplateBlock(name="generic_part") */
message messages[NUMBER_OF_PROCESSES];

byte token;

inline begin_round() {
    (token == _pid)
}

inline end_round() {
    token--
}

inline send_to_all(_message) {
    messages[_pid-1].value = _message.value
}

inline wait_to_receive() {
    token--;
    (token == _pid)
}

inline receive(_message, id) {
    _message.value = messages[id].value
}

proctype Process() {
    message _message;
    byte i, j, k, l;
    do
    :: begin_round();
        compute_message(_message);
        send_to_all(_message);
        wait_to_receive();
        state_transition();
        end_round()
    od
}

init {
    byte i, j, remaining_asynchronous_rounds =
        NUMBER_OF_ASYNCHRONOUS_ROUNDS;
    bool synchronous;

    system_init();

    atomic {
        for(i : 1..(NUMBER_OF_PROCESSES)) {
            run Process()
        }
    }

    do
    :: (token == 0);
        system_every_round();
        token = NUMBER_OF_PROCESSES;
        (token == 0);
        token = NUMBER_OF_PROCESSES
    od
}
/* @EndTemplateBlock */
```

When submitting the above PROMELA code as input for PROMNeT++, several header/source files are produces. Translated code relative to the *Process* nodes is stores in a file named *_process.cc*, whose code (in this case) is as follows:

```
#include "_process.h"
#include "types.h"
#include "utilities.h"

#include "message_m.h"
#include <omnetpp.h>

extern int rnd;
extern int round_id;
extern process_state state[NUMBER_OF_PROCESSES];


void Process::initialize() {
    ProcessInterface::initialize();
    _pid = getIndex() + 1;
    received_message_count = 0;
    //Variable initialization
    i = 0;
    j = 0;
    k = 0;
    l = 0;

}

void Process::handleMessage(cMessage* msg) {
    if (msg->isSelfMessage()) {
        /*
         * Messages whose class is "Message" are messages that a
         * process sent to itself. If we encounter these, we must
         * queue them, as if any other process sent them.
         */
        if (strcmp(msg->getClassName(), "Message") == 0) {
            enqueue_message(msg);
        }
        else if (strcmp(current_location, "main") == 0) {
            int step = step_map["main"];
            if (step == 0) {
                begin_round();
            }
            if (step == 1) {
                compute_message(this->_message);
                send_to_all(this->_message);
            }
            if (step == 2) {
                state_transition();
                end_round();
            }
        }
    } else {
        const char* sender_name = msg->getSenderModule()->
            getName();
        //Messages from init process
        if (strcmp(sender_name, "init") == 0) {
            //"init" message
            if (strcmp(msg->getName(), "init") == 0) {
                //Placeholder
            }
            //"new_round" message
            else if (strcmp(msg->getName(), "new_round") == 0) {
                step_map["main"] = 0;
                scheduleAt(simTime(), empty_message);
            }
            //"begin" message
            else if (strcmp(msg->getName(), "begin") == 0) {
                step_map["main"] = 1;
                scheduleAt(simTime(), empty_message);
            }
            delete msg;
        }
        //Messages from another process
        else if (strcmp(sender_name, "process") == 0) {
            enqueue_message(msg);
        }
    }
}

void Process::finish() {
    ProcessInterface::finish();
```

```
}

void Process::enqueue_message(cMessage* msg) {
    Message* message = check_and_cast<Message*>(msg);
    Process* sender = check_and_cast<Process*>(
        message->getSenderModule());
    int sender_pid = sender->_pid;
    received_messages[sender_pid - 1] = message;
    ++received_message_count;
    if (received_message_count == NUMBER_OF_PROCESSES) {
        ++step_map["main"];
        scheduleAt(simTime(), empty_message);
    }
}

//Generic functions
void Process::begin_round() {
    send(ready_message->dup(), "init_gate$o");
}

void Process::end_round() {
    for (int i = 0; i < NUMBER_OF_PROCESSES; ++i) {
        delete received_messages[i];
    }
    received_message_count = 0;
    send(finished_message->dup(), "init_gate$o");
}

void Process::send_to_all(message_t& msg) {
    Message* message = new Message();
    message->set_message(msg);
    for (i = 1; i <= NUMBER_OF_PROCESSES; ++i) {
        if (i == _pid) {
            scheduleAt(simTime(), message->dup());
        } else {
            send(message->dup(), "process_gate$o", i);
        }
    }
    delete message;
}

void Process::receive(message_t& msg, byte id) {
    Message* message = received_messages[id];
    msg = message->get_message();
}

//Specific functions
void Process::compute_message(message_t& _message) {
    _message.value = my_state.local_value;
```

```
}

void Process::state_transition() {
    my_state.received_message_count = 0;
    for (i = 0; i <= (NUMBER_OF_PROCESSES - 1); ++i) {
        if (my_state.received_message[i]) {
            receive(_message, i);
            my_state.values[i] = _message.value;
            my_state.received_message_count++;
        }
        else {
            //Skip
        }
    }
    if (my_state.received_message_count > (2 *
        NUMBER_OF_PROCESSES / 3)) {
        l = 0;
        for (i = 1; i <= (NUMBER_OF_PROCESSES); ++i) {
            k = 0;
            for (j = 0; j <= (NUMBER_OF_PROCESSES - 1); ++j) {
                if (my_state.values[j] == i) {
                    k++;
                }
                else {
                    //Skip
                }
            }
            if (k > l) {
                my_state.local_value = i;
                l = k;
            }
            else {
                //Skip
            }
        }
    }
    else {
        //Skip
    }
    if (l > (2 * NUMBER_OF_PROCESSES / 3)) {
        ASSERT((my_state.decision_value == my_state.local_value)
            || (my_state.decision_value == 0));
        my_state.decision_value = my_state.local_value;
        utilities::printf(this, "MSC: P%d decides %d on round
            %d\n", _pid, my_state.decision_value, round_id);
    }
    else {
        //Skip
    }
```

```
}
```

B.1 The translation process, explained

Like the vast majority of C++ source code, *_process.cc* starts with the inclusion of the appropriate header files. A few external variable declarations follow, as they are variables that are initializes in other source code files. We are then presented with OMNeT++'s basic routines: *initialize*, *handleMessage* and *finish*. Every node in OMNeT++ must have these routines, by design: *initialize* performs all necessary operations for initializing the node's internal state; we can clearly see that every process initializes its own PID (process ID), and an extra variable for counting how many messages that process has received, as well as its own PROMELA byte variables, *i, j, k, l*. *handleMessage* dictates what should happen when this process has received a message, whether its a message it sent to itself (via *scheduleAt*), or a message from another process (including the *init* process). Finally, *finish* performs the necessary clean-up operations when a simulation is terminated.

We then notice there's yet another function that doesn't originate in the PROMELA model, but is necessary nonetheless. *enqueue_message* stores any incoming messages for this process, after they've been sent, in a message array ("received_messages"). This allows processes to call the *receive* routine later, without blocking.

The PROMELA routines then follow. They're divided, as the comments indicate, into "generic functions" and "specific functions". Generic functions are PROMELA routines that belong to the PROMELA model's generic part (enclosed within the
@BeginTemplateBlock(name="generic_part") and *@EndTemplateBlock*
annotations in the model).

Notice, however, that none of the generic functions are direct translations from their PROMELA counterparts. This is due to, in part, the usage of OMNeT++'s API, and the presence of a different synchronization mechanism between rounds. The PROMELA implementation of OneThirdRule, as shown above, synchronizes processes via token passing. The corresponding implementation in OMNeT++, on the other hand, uses simple messages such as "new_round", "begin", "ready", and "finished", exchanged between the *init* process and the *Process* processes:

1. The *init* process sends a "new_round" message to all other processes, indicating the beginning of a new round.
2. The other processes reply to *init* with a "ready" message, indicating they're ready to commence the round.
3. *init* replies with a "begin" message, instructing all processes to perform their operations.
4. When the round is over, all processes (excluding init) clear their internal message array, send a "finished" message to *init*, and await for the next "new_round" message.
5. Repeat from step 1 above.

Finally, the specific functions are directly translated to C++ code, and are nearly identical to their PROMELA counterparts. Notably, PROMELA's *d_step* block is omitted, as it's only relevant for simulating the PROMELA model via Spin. *if* blocks are translated here as C++ *if-else* statements (a common pattern here is "if, else skip"). Likewise, *for* loops in PROMELA are directly mapped to their suitable C++ implementations; a loop in the form *for(variable : START..END)* is translated as *for (variable = START; variable <END; ++variable)* in C++.