MSc Thesis

Master's Programme in Computer Science

Modelling a Distributed Data Acquisition System

John Lång

June 7, 2021

Faculty of Science

University of Helsinki

Supervisor(s)

Prof. Keijo Heljanko

Examiner(s)

Prof. Keijo Heljanko,

Ph.D. Jarno Alanko

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)

00014 University of Helsinki, Finland

Email address: info@cs.helsinki.fi

URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Master's Programme in Computer Science

Tekijä — Författare — Author

John Lång

Työn nimi — Arbetets titel — Title

Modelling a Distributed Data Acquisition System

Ohjaajat — Handledare — Supervisors

Prof. Keijo Heljanko

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
MSc Thesis	June 7, 2021	72 pages, 3-page appendix

Tiivistelmä — Referat — Abstract

This thesis discusses the formal modelling and verification of certain non-real-time aspects of correctness of a mission-critical distributed software system known as the ALICE Data Point Service (ADAPOS). The domain of this distributed system is data acquisition from a particle detector control system in experimental high energy particle physics research. ADAPOS is part of the upgrade effort of A Large Ion Collider Experiment (ALICE) at the European Organisation for Nuclear Research (CERN), near Geneva in France/Switzerland, for the third run of the Large Hadron Collider (LHC). ADAPOS is based on the publicly available ALICE Data Point Processing (ADAPRO) C++14 framework and works within the free and open source GNU/Linux ecosystem.

The model checker Spin was chosen for modelling and verifying ADAPOS. The model focuses on the general specification of ADAPOS. It includes ADAPOS processes, a load generator process, and rudimentary interpretations for the network protocols used between the processes. For experimenting with different interpretations of the underlying network protocols and also for coping with the state space explosion problem, eight variants of the model were developed and studied. Nine Linear Temporal Logic (LTL) properties were defined for all those variants.

Large numbers of states were covered during model checking even though the model turned out to have a reachable state space too large to fully exhaust. No counter-examples were found to safety properties. A significant amount of evidence hinting that ADAPOS seems to be safe, was obtained. Liveness properties and implementation-level verification among other possible research directions remain open.

ACM Computing Classification System (CCS)

Theory of computation → Logic → Verification by model checking
Computer systems organisation → Architectures → Client-server architectures

Avainsanat — Nyckelord — Keywords

distributed systems, control systems, data acquisition, formal verification, model checking, case study

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — Övriga uppgifter — Additional information

Algorithms study track

Contents

1 Introduction **1**

2 Background **3**
- 2.1 The O^2 Project . . . 3
- 2.2 ALICE Data Point Service . . . 4
- 2.3 Requirements of ADAPOS . . . 6
- 2.4 What Makes Testing Insufficient for ADAPOS? . . . 9
- 2.5 Why Formal Methods? . . . 11
- 2.6 Why Model Checking in Particular? . . . 12

3 Enumerative LTL Model Checking **15**
- 3.1 Finite State Machines . . . 16
- 3.2 Linear Temporal Logic . . . 21
- 3.3 Enumerative Model Checking . . . 23
- 3.4 State Space Explosion . . . 25

4 The Model of ADAPOS **30**
- 4.1 Processes in the Model . . . 33
- 4.2 Differences Between the Model and the Implementation . . . 34
- 4.3 The Base Variant . . . 37
- 4.4 Input Counters Variant . . . 43
- 4.5 Direct Input Variant . . . 43
- 4.6 Direct Input Counters Variant . . . 44
- 4.7 Structured Input Variant . . . 44
- 4.8 Structured Input and Output Variant . . . 46
- 4.9 Input Channels Variant . . . 47
- 4.10 Input and Output Channels Variant . . . 48
- 4.11 Spin to LaTeX Converter . . . 48

5 The Specification of ADAPOS **50**
- 5.1 Sanity . . . 50
- 5.2 Safety . . . 51
- 5.3 Disjointness . . . 52
- 5.4 Checksums . . . 52
- 5.5 Causality . . . 52
- 5.6 Monotonicity . . . 53
- 5.7 Combined Effect of the Safety Properties . . . 54

6 Results **56**

7 Discussion **60**
7.1 Lessons Learned . . . 60
7.2 Experiences with Tools and Technologies . . . 60
7.3 Related Work . . . 61
7.4 Conclusion . . . 62
7.5 Future Prospects . . . 63
7.6 Acknowledgements . . . 64

References **64**

A Derivation for Sanity in the Default Mode **i**

1 Introduction

A particle detector known as *a Large Ion Collider Experiment* (ALICE)[1] is one of the eight experiments of the *Large Hadron Collider* (LHC)[2] at *European Organisation for Nuclear Research* (CERN)[3] near Geneva in Switzerland and France. ALICE specializes in measuring properties of quark-gluon-plasma, an extremely hot and dense form of matter, produced in so-called heavy ion collisions involving two lead ions or a lead ion and a proton, accelerated to ultra-relativistic speeds. This research is important for building an understanding of the conditions of the early universe, just moments after the Big Bang.

Apart from the exciting prospects in physics and cosmology, the LHC experiments, including ALICE, also present formidable challenges to engineers and computer scientists. In terms of number of subsystems and experts of different fields involved, the complexity of LHC experiments is comparable to spaceship construction. According to [1, Table 4.1], the expected rate of raw physics data produced by the detectors of ALICE during the third run (RUN3) of LHC that starts in 2021[4] is more than a terabyte per second. Compared to the situation during the second run, the amount of physics data obtained in ALICE is going to grow hundred-fold.

In order to cope with the increase in data throughput, the data delivery architecture of ALICE from detectors all the way to the LHC Grid computing cloud needs to be upgraded. *ALICE Data Point Service* [48] (ADAPOS) is a distributed system that is part of this upgrade effort. It moves a subset of the data produced by subdetectors of ALICE towards the LHC Grid computing cloud for particle track reconstruction and physics analysis. The article [48, p. 482] lists five requirements for ADAPOS:

1. It must not lose or corrupt data;
2. it must preserve the ordering of data elements;
3. its throughput and latency must be as predictable as possible;
4. it must be stable and responsive; and
5. its processes must have redundant instances for maximising availability and making maintenance easier.

Originally, unit and integration tests were used for finding defects in ADAPOS implementation. Testing alone is insufficient for demonstrating the correctness of a system such as ADAPOS. The great Edsger Dijkstra has famously said [10, p. 21]:

[1] https://home.cern/science/experiments/alice
[2] https://home.cern/science/accelerators/large-hadron-collider
[3] https://home.cern/
[4] https://lhc-commissioning.web.cern.ch/schedule/LHC-long-term.htm

"Testing shows the presence, not the absence of bugs."

The heart of the problem is that ADAPOS is a distributed system made of concurrently executing multi-threaded processes running on separate machines connected by network. There are simply too many possible scenarios to capture into hand-written test cases. The cppunit[5] test framework used for developing ADAPOS does not control scheduling of threads. When threads access shared memory, there can be data races that have different outcomes depending on scheduling. Unpredictable resolution of data races may translate into *flaky* test cases which sometimes succeed and sometimes fail. Reproducing rare system behaviours with flaky tests can be hard.

A portion of software systems are critical in some sense. According to [31, Table 2], a system is *mission-critical* if a failure

> "[m]ay lead to an inability to complete the overall system or project objectives; e.g., loss of critical infrastructure or data."

The failure of ADAPOS could ruin the analysis of physics data collected from the subdetectors, making ALICE effectively a 10,000-tonne paperweight with huge electricity consumption. Thus, ADAPOS qualifies as a mission-critical system. Furthermore, stopping expensive experiments for patching ADAPOS is not on the table. These factors justify investing more than the usual amount of effort into verification and validation of ADAPOS.

The shortcomings in testing and the importance of getting ADAPOS work right led the author of this thesis to consider formal verification. Based on the good experiences obtained with a related system [49], model checking was chosen as the method for performing the current study. Simply put, *model checking* [2, 17, 38, 58] is the process of building a mathematical model of the system under verification and algorithmically checking if the model satisfies a specification written in formal logic.

The goal of this study is to demonstrate the correctness of ADAPOS with respect to the properties refined from the five requirements above. There are three levels of scope which contribute to the overall correctness of ADAPOS:

1. The part of the underlying C++ software framework that may run persistently;
2. The individual component processes of ADAPOS; and
3. ADAPOS as a distributed system.

Level 1. has been mostly verified [49]. The focus of the current work is on levels 3. Level 2 had to be left out as future work. Chapter 2 provides more context and motivation. Chapter 3 introduces the theory behind the methods used. Chapter 4 describes the abstract model that was developed. Chapter 5 presents the specification of the model. Chapter 6 summarises the results. Finally, Chapter 7 closes this thesis with discussion.

[5] https://sourceforge.net/projects/cppunit/

2 Background

As mentioned above, for verifying that ADAPOS meets its requirements, a formal verification technique known as model checking was chosen for this study. The model checker chosen for model checking the abstract specification of ADAPOS was Spin [34]. Another model checker, DIVINE [4] was tried for model checking the C++ implementation of ADAPOS. Implementation-level modelling had to be dropped due to technical issues with DIVINE and for keeping the scope of this work suitable for a master's thesis. This chapter provides background and motivation for the research question and the method chosen for answering it.

2.1 The O^2 Project

Until the third run of LHC, data from ALICE was processed in two stages. *Online* processing decides which measurements need to be preserved for further analysis and which measurements can be discarded as noise or otherwise uninteresting data. *Offline* processing reconstructs particle tracks from the data filtered online and compares them with predictions made by theoretical models. The O^2 project, whose name refers to the conjunction of offline and online, aims to combine these two currently separate stages of processing into one real-time computation pipeline.

The ALICE experiment produces data relevant to O^2 in two categories: physics data and conditions data. *Physics data* includes the readings from the numerous subdetectors of ALICE, e.g., gas detectors, silicon detectors, or scintillating crystals. Physics data forms the vast majority of the data produced by volume. The ALICE *Data Acquisition* (DAQ) system is responsible for collecting physics data and delivering it to Grid.

Conditions data, on the other hand, is data on the subdetectors themselves and their supporting infrastructure. Conditions data records the ambient conditions of ALICE. For example, the output voltages of power sources or gas temperature or pressure readings in gas detectors fall into this category.

Conditions data are crucial for maintaining optimal operating conditions for detectors. For example, too high input voltage can break the very delicate and expensive silicon detectors. For many of the devices, there are no commercially available replacements. The task of the ALICE *Detector Control System* (DCS) [14, 15] is to maximise the duration of optimal working conditions for physics detectors by using conditions data and possible commands from human operators.

Some of the physics calculations involved in particle track reconstruction are sensitive to ambient conditions. Thus, conditions data has a second role in providing correcting coefficients to the physics calculations. DCS and DAQ are designed to be two independent systems separated by a firewall, so DAQ cannot directly acquire conditions data from DCS.

2.2 ALICE Data Point Service

ADAPOS contributes to the O^2 project. The purpose of ADAPOS is to transmit conditions data from DCS to DAQ. It succeeds an application known as Amanda which is part of the ALICE offline Shuttle [68] system. Figure 1 gives an overview of ADAPOS.

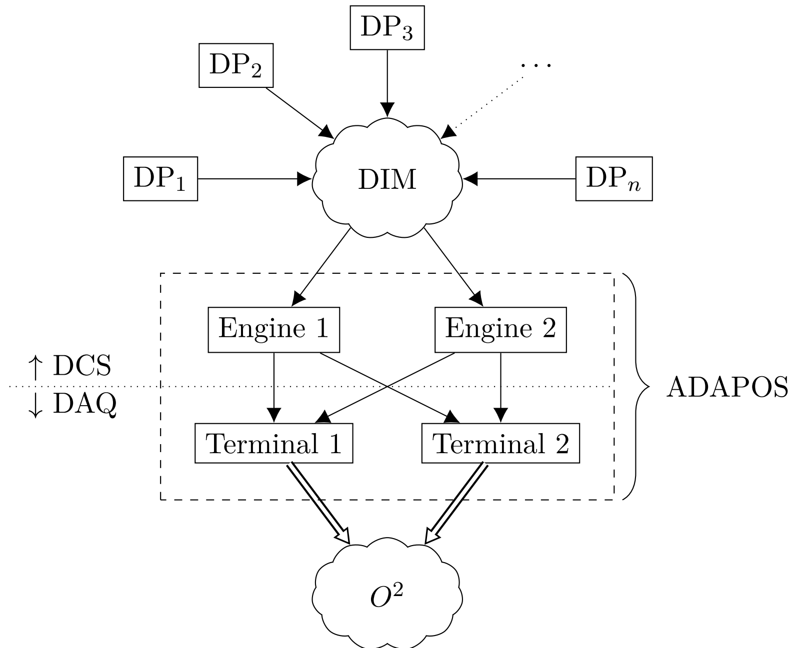

Figure 1: A simplified presentation of ADAPOS deployment.

The conditions data in ALICE DCS consists of data points. A *data point* represents the state of (a channel in) a hardware device, or some other logical or aggregated piece of information. There is separation of concerns between the DCS and the DAQ subsystems. The *service* delivering data points has been split into two types of processes accordingly: Engines operating on DCS servers and Terminals running on the DAQ side.

All data are replicated between two redundant copies of Engines and Terminals in order to increase the availability of the service. Engines and Terminals maintain indexed data structures known as *Full Buffer Images* (FBIs). An FBI represents a snapshot of the state of all data points published through ADAPOS. The number of data points is fixed; if it changes, then ADAPOS has to be reconfigured and restarted.

Data points are published to ADAPOS using the *Distributed Information Management* (DIM) [25] protocol, following a publisher-subscriber pattern with a service broker known as a DIM DNS server. DIM is developed and

maintained at CERN and is freely available[6]. Another protocol, called 0MQ[7], is used for transmitting data points between Engines and Terminals and also between Terminals and DAQ readout software.

In Figure 1, The bold arrow between Terminals and O^2 emphasises the fact that the FBIs are sent as discrete messages of dozen or so megabytes in size. By contrast, other arrows denote continuous event-based data streams containing only changes to individual data points that arrive as they are measured. DP stands for a data point.

During the start-up of ADAPOS, Engine queries the DIM DNS server which returns the addresses of the servers publishing the data points. After that, Engine establishes a direct connection with the relevant servers. Figure 2 presents the interaction between an Engine and a Terminal after Engine has started.

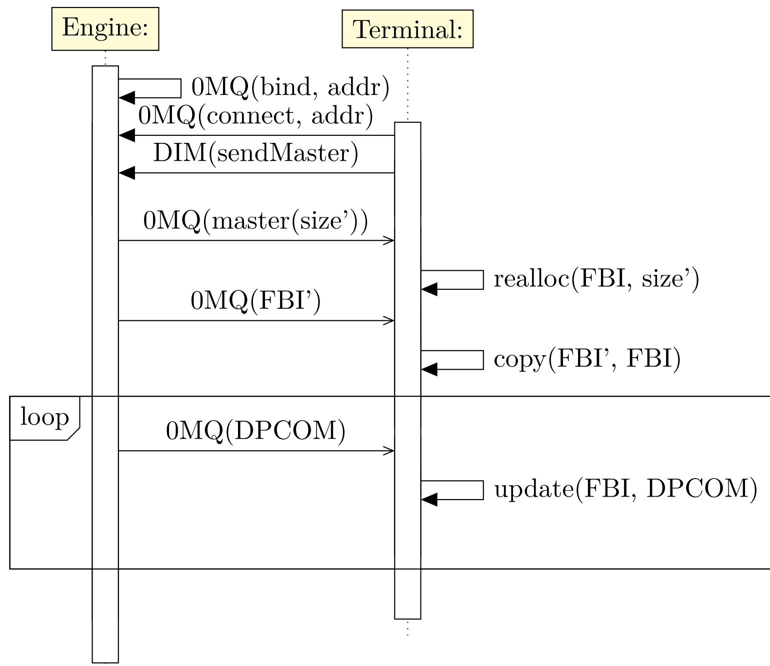

Figure 2: A UML sequence diagram showing the key events in the communication between a Terminal and an Engine.

After the start-up phase, whenever the state of a data point changes, the new value is transmitted to Engine. Engine then sends a fixed-size binary record known as a *Data Point Composite Object* (DPCOM) to Terminal over a 0MQ socket.[8] A DPCOM contains the value of a single data point along

[6] http://dim.web.cern.ch/
[7] https://zeromq.org/
[8] See Chapter 4 of the ADAPRO manual for the classes used in ADAPOS at https://gitlab.com/jllang/adapro/-/jobs/artifacts/5.0.0/download?job=manual

with the name of the device or channel it represents and other metadata such as the type of the data point, a timestamp, and bit flags.

ADAPOS applications have been implemented on top of the open source C++ software framework, the *ALICE Data Point Processing Framework* (ADAPRO)[9]. Inspired partly by the SMI++ [23][10] framework used in control systems at CERN, ADAPRO follows a state machine model which has been formally verified for certain safety and liveness properties [49]. The C++ implementation of the state machine mechanism was also partially verified in the same study. Figure 3 presents the technological ecosystem supporting ADAPOS.

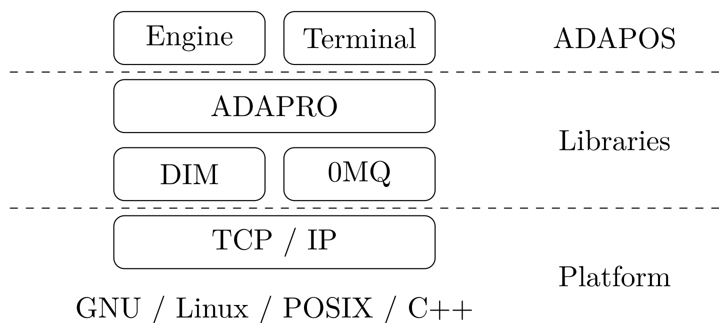

Figure 3: The technological ecosystem which ADAPOS inhabits.

2.3 Requirements of ADAPOS

This chapter discusses the specification of ADAPOS using the English language. The specification will be formalised in Chapter 5. The properties investigated in this thesis were developed from earlier requirements [48, p. 482] and discussed above in Chapter 1.[11] Unfortunately, due to known theorems [21, 27] it is impossible to complete all of these goal to be listed below perfectly in the same system. ADAPOS aims to find a sensible compromise. The main properties investigated in this thesis are listed below.

1. *Atomicity*: ADAPOS must never produce output containing partial or incomplete data points.

2. *Disjointness*: ADAPOS must never mix updates to separate data points. Specifically, all data points must be stored in non-overlapping locations.

[9] https://gitlab.com/jllang/adapro

[10] https://smi.web.cern.ch/

[11] See also the manual of ADAPOS Engine at https://gitlab.com/jllang/adapos-engine/-/jobs/713674023/artifacts/download

3. *Causality*: ADAPOS must never produce output values that it did not receive as input.

4. *Monotonicity*: ADAPOS must never change the order of updates to data points.

5. *Availability*: There should always be an Engine-Terminal pair available that is able to deliver up-to-date data points.

A software system may have *safety* [2, p. 109] and *liveness* [2, p. 119] properties. According to Leslie Lamport, safety properties state that "something will not happen" [44, p. 125] while liveness properties state that "something must happen" [44, p. 125]. Safety and liveness properties can also be distinguished by how they are refuted.

Safety properties are those properties that have counter-examples of finite length. Type safety (e.g., that variables declared having type `int` can never have `string` type values) and absence of division by zero, illegal memory accesses, or deadlocks are examples of safety properties. A stack trace that is printed when a Java program halts abnormally due to an unhandled `NullPointerException`, is a counter-example to the safety property asserting that the program never dereferences the `null` reference.

Liveness properties have counter-examples that are infinitely long. Depending on the property, a counter-example may never satisfy it or satisfy it at most finitely many times. Halting is a commonly needed liveness property. A counter-example to halting is an infinite, i.e. never-ending execution. Not all programs are meant to halt, though. In fact, ADAPOS processes should only halt in case of a scheduled maintenance. Livelocks are also typical liveness violations.

Properties 1 – 4 are safety properties. A natural desirable safety property would be that ADAPOS must never alter the data it transmits. This is related to the concept of data independence introduced by Pierre Wolper [67]. Considering the modelling and verification effort involved, and the use case at CERN, properties 1 – 4 together give reasonable guarantees for ADAPOS.

ADAPOS can be used for transmitting arbitrary binary data without type information. It would be hard to define and verify safety properties stronger than properties 1 – 4 in the general case. Furthermore, these four properties allow dropping or duplicating data points, which gives the implementation some slack when the network is highly congested or some data points are updated infrequently.

In addition to the safety properties, it is important to ensure the *availability* of ADAPOS, which is a liveness property. Availability means that at any given moment of time, there is at least one Engine-Terminal pair able to deliver conditions data to O^2. As mentioned earlier, maximising availability is the reason why ADAPOS has redundant copies of the Engine and Terminal processes.

There are interesting properties that had to be excluded from this study. Updates to certain data points may arrive scarcely. It is all the more important that they get through. The property that all updates eventually arrive at O^2 would be *reliability*.

Ensuring reliability would require storing updates in a buffer that could theoretically grow without a bound. If ADAPOS processes ran out of their finite amount of memory, they would cease operation, violating availability. An unbounded memory requirement would also preclude certain formal verification methods, including model checking. Furthermore, arbitrarily long buffers would also introduce unbounded latency. Trade-offs between conflicting requirements are inevitable.

ADAPOS processes use a bounded amount of memory to avoid running out of resources or introducing unnecessary latency. In the previous study [48, p. 484], no evidence of packet loss was found. Considering the effort invested in simulating ADAPOS under maximum load, and the fact that no event losses have ever been observed, it can be argued that packet loss is unlikely to occur under foreseeable operating conditions. As mentioned in the previous study [48, p. 485],

> "[t]he maximum level of performance with the test setup [of ADAPOS] was found to be around 400 kHz, which is two orders of magnitude better than the current requirements."

400 kHz means 400,000 data points per second.

Still, with the current design of ADAPOS, dropping updates under force majeure conditions cannot be ruled out theoretically. It is generally impossible to guarantee availability and atomic consistency (a strong form of reliability) in the same distributed system [27, p. 53]. ADAPOS prefers availability over reliability.

An important property is that if ADAPOS drops updates, it must do it without introducing biases to the conditions data. This property, *neutrality* means that no particular set of data points can be favoured over others by the architecture of ADAPOS. Neutrality would be very expensive to verify using model checking, because neutrality requires strong fairness assumptions for every data point separately.

A liveness property that was considered and even formalised, was called *synchrony*. This property asserts that every once in a while, all Terminal processes must have the same data points in their FBIs. However, this property was considered too demanding to be guaranteed in all possible situations, so it was dropped.

Another notion of synchrony, or *consensus* more precisely, is about deciding which Engine-Terminal pair serves as the active route for transmitting the conditions data and making sure that all processes involved agree on the decision. For a distributed system featuring replicated processes, it is

important to be able establish a consensus on the roles of each process. There are distributed consensus algorithms such as Paxos [46] and Raft [53].

An arrangement to use the distributed file system ETCD[12] to keep track on the active Engine-Terminal pair was considered. In principle, all of conditions data could be handled in a message passing system such as Apache Kafka[13], which would also take care of consensus problems. Such a huge design change would be too late at this point considering the schedule of RUN3. The evidence on the more than satisfactory performance [48] of ADAPOS also argues against such a redesign, as all the hard work on testing and measuring (in addition to designing and implementing) the system would need to be redone.

It was decided to keep all Engines and Terminals always active and agnostic about processes outside ADAPOS accessing conditions data. In a way, this design decision moved the consensus problem to O^2. Now the recipient of the data streams has to decide which Terminal to connect to and there are no guarantees that all Terminals have the same data.

The last property to mention is that conditions data should be up-to-date in the sense that ADAPOS adds a *bounded amount of latency* to data point updates passing through it. This is related to the requirement 3 listed in Chapter 1. This requirement has been taken into account in the C++ implementation. Real-time properties such as bounded latency are out of the scope of this thesis, however. The model checker UPPAAL[14], has been used for verifying real-time systems, e.g., in nuclear power domain [43].

Preliminary latency measurements were made during the early simulations [48], but they were considered to be too early for publication at that point, as details of the design and deployment of ADAPOS would change. Latency was not considered to be a critical concern of the design of ADAPOS. Much of the conditions data arriving at ADAPOS Engine is already relatively old and ADAPOS usually adds only milliseconds or so to the overall latency. For example, some power supply devices read out their channel voltages twice per minute. Furthermore, ALICE DCS applies smoothing to the conditions data [15, Section III]. The distance between devices and ADAPOS Engine (or Terminal) in terms of the number of different machines involved in transport and processing is quite long.

2.4 What Makes Testing Insufficient for ADAPOS?

The usual approach to verifying the correctness of a piece of software is to write tests to see how the system reacts to a handful of possible inputs. In this paradigm, it is typically the interactions between the software system and its environment or tests, as well as the feedback from the clients or

[12] https://etcd.io/

[13] https://kafka.apache.org/

[14] https://www.it.uu.se/research/group/darts/uppaal/examples.shtml

end-users, that define what correctness means. ADAPRO and ADAPOS originally relied on tests and hundreds of hours of simulations [48]. This work was certainly valuable but it did not remove all reasonable doubts on the correctness of the specification and design of ADAPOS.

The main difficulty in testing ADAPRO and ADAPOS were the complex concurrent behaviours arising from interactions between threads or processes. Concurrency-related defects were hard to anticipate in the informal design process and to reproduce with tests. Chapters 5 and 6 of the article [49] discuss four defects that were found in the formal verification of ADAPRO. Model checkers were able to quickly uncover these issues whereas hundreds of hours of testing and simulation had failed to do so.

Generally speaking, there are too many behaviours to test individually for achieving complete or even relatively high coverage. Exhaustive testing of a concurrent system takes double exponential amount of work with respect to the number of parallel processes, as noted by Petr Ročkai in his dissertation [59, Footnote 1 on p. 10]. Exhaustive testing is thus challenging.

One consideration in choosing verification methods is the level of risks involved in the project. Wikipedia lists cautionary tales on what may happen if verification is not done carefully enough[15]. The sheer number of software defects with catastrophic consequences in the past provides motivation for investing effort into preventing future incidents.

The failure of ADAPOS would mean that either O^2 gets no conditions data at all, or the data is stale or incorrect. This in turn would hinder the particle track reconstruction process. If the physics data cannot be analysed accurately, then there is no point in running an expensive physics experiment. Even though it is a small system, ADAPOS is a link in a chain of critical software components of the ALICE experiment. The ADAPOS project aims not to be the weakest link in that chain.

From a business point of view, as conjectured by Gerard Holzmann [35, p. 81], the problems most likely to be exposed by testing may not be those that pose the greatest risk. One study goes even further to note that testing can even miss "simple, obvious faults than one may expect" [41, p. 217]. ADAPOS qualifies as a mission-critical system with economically high stakes. This motivates the search for high-risk defects in addition to the easily testable ones.

If testing cannot be carried out exhaustively, then the question is how to direct the testing effort in a way that maximises the chance of finding bugs. Methods such as Whitebox fuzzing [29], directed automated random testing [28], symbolic execution [11], and concolic testing [63] aim to increase test coverage by automatically generating inputs to tests. Regardless, earlier positive experiences with model checking [49] made the author of this thesis choose model checking over these advanced forms of testing.

[15] https://en.wikipedia.org/wiki/List_of_software_bugs

2.5 Why Formal Methods?

Formal methods focus on demonstrating (or finding counter-evidence against) the correctness of a piece of software by employing mathematical techniques. The range of techniques under the umbrella of formal methods is quite extensive. It is thus hard to give a precise definition for the term. Christel Baier and Joost-Pieter Katoen characterise formal methods as "the applied mathematics for modeling and analyzing [Information and Communication Technology] systems" [2, p. 7].

Testing, static analysis, model checking, and theorem proving form a continuous spectrum of methods, each one choosing different trade-offs between exhaustiveness, precision, speed of execution and ease of use. At least a part or an aspect of the system under verification must be unambiguously defined in mathematical terms in order to apply formal methods. That formal definition can sometimes be generated or extracted programmatically. The formal specification of ADAPOS was written by hand, because it was easier to do that way.

There are numerous case studies and success stories on using formal methods in software development [22]. A seminar report[16] written by the author of this thesis lists a few reported uses of model checking for verifying real life systems and elaborates three of them in detail. In addition to the ADAPRO case [49], some interesting examples from the past decade or so include the Address Resolution Protocol [9], slats and flaps control unit of an aircraft [13], attitude and orbit control system of a satellite [24], the control system of another LHC experiment, the Compact Muon Solenoid control system [37], a stepwise shutdown system and uninterruptible power supply control software in nuclear power domain [43], a compiler for a subset of the C programming language [47], an operating system microkernel [41], just to name a few.

The first chapter in the book by Baier and Katoen [2] mentions some real-life examples of software defects exposed with model checking. The website of the Spin[17] model checker lists important use cases of the tool. There is another list of Spin case studies online.[18] Other model checking and formal verification tools often list case studies and other research articles on their own websites.[19]

Apart from technical arguments, economical considerations often support formal methods, too. Gerard Holzmann [35] and John Fitzgerald et al. [22] provide business-oriented argumentation on why formal methods are worth using. Jonathan Bowen and Michael Hinchey discuss some com-

[16] https://www.cs.helsinki.fi/u/jllang/Three_Examples_on_the_Real_Life_Applications_of_Model_Checking.pdf

[17] https://spinroot.com/spin/success.html

[18] http://www.imm.dtu.dk/~albl/promela.html

[19] E.g., https://ti.arc.nasa.gov/tech/rse/publications/vnv/#model

mon preconceptions and provide general guidelines on the use of formal methods [8].

The rise of specialised companies relying on formal methods, such as Coverity [7] (which was acquired by Synopsys in 2014), Galois [55] and IOHK [40], hint that there might be a rising trend on the adoption of formal methods in the software industry. Large established companies such as Amazon [51], Microsoft [3], and former Lucent [12] (which was later acquired by Nokia) are also known for using formal methods to their profit. In the computer hardware industry, AMD [60], Centaur [30] and Intel [39] for example, have successfully deployed formal verification in their processes.

The case studies and companies mentioned above are just select few examples. More thorough overview on different flavours of formal verification and their industrial adoption is worthy of a survey article or a thesis in its own right. At any rate, the there is a growing corpus of literature demonstrating that formal methods are not merely an academic exercise that can be easily dismissed. There are industrial segments relying on formal methods.

2.6 Why Model Checking in Particular?

ADAPOS is a distributed system made of reactive components. Apart from possible bugfixes and design changes necessitated by the findings made during the verification, ADAPOS is more or less complete, for the time being. Since the source code already exists, a correct-by-construction approach (e.g., by constructing the system using a framework like Verdi [66] or directly with an interactive proof assistant [57] like how the CompCert C compiler [47] was made) is no longer attractive. Neither is a refinement approach such as the one taken in the seL4 case [41].

According to Edmund Clarke et al. [17, p. 4], model checking in general has three advantages:

- Ideally, model checking can be fully automated without need for verification experts in software development teams.
- Model checking can be applied at different stages and abstraction levels of software development processes, from abstract designs to concrete program code.
- Model checking is particularly suitable for analysing concurrent systems.

Baier and Katoen [2, p. 14-16] list eight other strong points of model checking:

- It is suitable to many applications such as embedded systems, software engineering, and hardware design.
- It supports partial verification; individual properties can be checked without having to supply a complete specification at once.

- It is not sensitive to the probability of an error occurring; this contrasts model checking with testing and simulation which usually find only the most easily exposed errors.

- It provides diagnostic information in case a property violation is detected which is useful for debugging.

- It is a potential "push-button" technology requiring neither a high degree of user interaction or high level of expertise.

- It enjoys an increasing interest by industry; many hardware manufacturers run their own verification labs and offer model checking-related jobs, there are also commercial model checkers available these days.

- It integrates well with current development processes; it does not have a steep learning curve, and empirical studies indicate that it may shorten the development times.

- It has a sound mathematical foundation based on the theory of graph algorithms, data structures, and logic.

All of the pros of model checking mentioned above are relevant for the work discussed in this thesis. That said, experience with the models studied in this thesis and in the previous article [49] have taught that especially the promises of a "push-button" technique can be a bit on the optimistic side. It can take weeks or months of work to have the model in a state where this metaphorical button can be finally pressed. Baier and Katoen [2, pp. 115–116] also list eight weaknesses of model checking:

- It is mainly suited to control-intensive applications; it is weaker on data-intensive applications that may have even infinite data domains.

- It depends on decision procedures; there are infinite-state systems and data types that require semi-decidable or even undecidable logics.

- It verifies a model of a system, and not the system itself; the results obtained are only as good as the model, so complementary techniques such as testing might still be needed.

- It checks only the properties suggested by the user; the validity of properties not checked remains open.

- It suffers from the state explosion problem (which will be discussed in Chapter 3.4); even the most advanced state space reduction techniques may not be able to make a model fit into a computer's memory.

- It requires a certain amount of expertise for finding the best abstractions for representing the system with a small model and stating the specification using mathematical logic.

- It may not deliver correct results if the model checker has a software defect, though parts of advanced model checking algorithms have been formally demonstrated to be correct using theorem provers.
- It does not work for generalisations: systems with an arbitrary number of components, i.e. parametrised systems, cannot be model checked; however, model checking may help to find hypotheses to be proved in a parametrised setting.

Based on personal experience, the author of this thesis would like to add to the list of shortcomings above that interpreting the counter-examples given by a model checker is not always easy. Clarke et al. [17, pp. 4 – 5] characterise three of the complementary methods with their pros and cons:

- *Testing*: There is a wide range of technologies for testing software, such as the familiar unit testing with hand-written inputs. Testing is generally the fastest and easiest software verification method to deploy. Due to its dynamic nature, testing can detect a wide range of errors, many of which are hard to find with purely static techniques. However, the weakness of testing is typically low coverage of state spaces, making it more suitable for debugging than verification. Advanced testing techniques, some of which were mentioned above at the end of Chapter 2.4, try to alleviate this shortcoming.
- *Abstract interpretation*: Though somewhat similar to model checking in some respects, abstract interpretation usually focuses more on speed while sacrificing some of its accuracy in exchange.[20]
- *Theorem proving* [57]: The disadvantage of this method is that it can be very time-consuming to the software developers, even more than model checking.

The above three families of methods are quite large and varied. Abstract interpretation itself is a general technique that can be used in many different settings, including model checking and theorem proving. All in all, based on the empirical observations and the more general reasoning above, it was decided that model checking would be a good method for carrying out the research in this thesis.

[20] Clarke et al. might be referring to the use of abstract interpretation in *static analysis* [52, Ch. 4].

3 Enumerative LTL Model Checking

Model checking [2, 16, 38, 59] is based on the idea of representing the system under verification as a mathematical model and its properties as logical formulae. The problem of deciding whether or not a system meets its specification becomes the problem of deciding if its model satisfies the formulae. A *model checker* implements a decision procedure for this satisfaction problem. Figure 4 presents the general idea of model checking schematically.

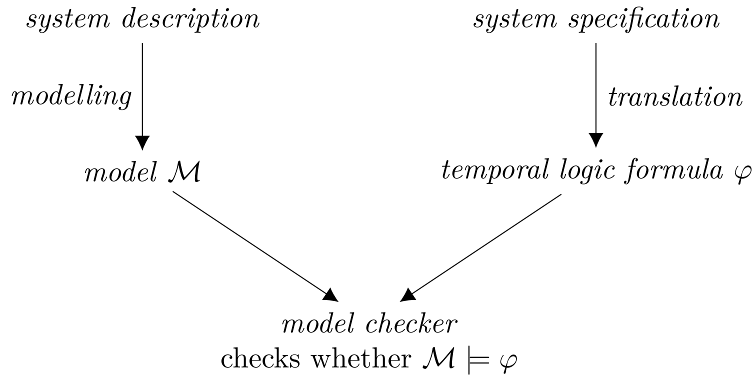

Figure 4: Outline of model checking. Adapted from [17, p. 3].

There are multiple approaches to model checking. As Clarke et al. [17, p. 5] put it,

> "... model checking is characterised less by purity of method than by the goal of debugging and analyzing dynamical systems that exist in the real world and can be modeled as state-transition systems."

Enumerative model checking [36] [38, Ch. 2] represents the model along with its properties explicitly as a finite state transition system. The satisfaction problem is solved by computing all the reachable states of the system and checking if they satisfy the property under investigation thus performing a *proof by exhaustion*. The model checkers DIVINE and SPIN used for the research work discussed in this thesis are enumerative.

Instead of relying on brute strength, industrial grade enumerative model checkers use optimisation techniques to reduce the search space. These techniques include partial order reduction [54] and bitstate hashing [33] used in SPIN, as well as τ-reduction [5] used in DIVINE. Optimisation techniques for model checking are outside the scope of this thesis, however, and will not be discussed further.

Even though model checkers are automated tools, understanding their internal mechanisms and general theory helps building models that can be optimised well by the model checkers. There are many formalisms for

representing finite-state systems such as (labelled) transition systems [2, p. 20] [54, p. 174], program graphs [2, p. 32] Kripke structures [17, p. 6] [42, p. 141], simple programs [38, Ch. 1.1], and various types of automata [26, p. 6] [36, p. 156] [42, p. 108] [61, Ch. 1]. These formalisms are all similar in spirit and differ only in specifics. There are likewise many different systems of formal logic used for specifying properties.

3.1 Finite State Machines

A *Finite State Machine* (FSM) [36, p. 156], also known as Finite (State) Automaton [42, p. 108] [61, pp. 51, 53], $(\Sigma, S, \iota, \delta, F)$ has

- a finite non-empty set Σ of symbols known as the *alphabet* (or label set or set of actions),
- a finite non-empty set S of states,
- an *initial state* $\iota \in S$ [36, p. 156] (which could equivalently defined to be a set of initial states [42, p. 108]),
- *transition* relation $\delta \subseteq S \times \Sigma \times S$, and
- a set $F \subseteq S$ of *final* (or accepting) states.

Following the convention of [2, p. 20], this text uses $s \xrightarrow{\alpha} s'$ to denote the proposition $(s, \alpha, s') \in \delta$.

A program or a model written in textual programming or modelling language source code can be converted into an FSM. The following naïve conversion was inspired by the operational semantics of NanoPromela [2, Ch. 2.2.5] and the transition system semantics of channel systems [2, Ch. 2.2.4] but simplified (omitting channels and guards) and adapted for FSMs. More rigorous semantics for modelling languages such as Promela [50] can be quite complex.

- The alphabet will be the set of all statements in the code.
- The states will be tuples $(\ell, \mathbf{v})$ containing the location ℓ of the current statement of code and the values $\mathbf{v} = v_0, v_1, \ldots, v_n$ of all the respective variables $x_0, x_1, \ldots, x_n$ occurring in the code. A special location `halt` can be used to mark those states for which there are no more statements left to execute.
- The initial state will point to the first statement and have suitable values (e.g., zeros) in all variables.
- There will be a transition $(\ell, \mathbf{v}) \xrightarrow{\alpha} (\ell', \mathbf{v}')$ if and only if

    1. ℓ is the location of the statement α;

2. ℓ' is the location arrived after executing α (usually the next line, but in case of control structures and `goto` statements it can be a different location); and
3. executing α changes the values $\mathbf{v}$ into the values $\mathbf{v}'$.

- All states can be made accepting. The reason for this choice will become clear later.

The conversion above tacitly assumes that the language of the code has well-defined operational semantics. Typically, this state-machine approach works best for imperative programs. For purely functional programs, other verification methods such as using the type system of the programming language to enforce safety invariants in a propositions-as-types [65] style and using equational reasoning might be better options.

Consider the lines labelled as ℓ_1 to ℓ_4 in the C language subroutine `fact` shown in figure 5. The code uses a `goto` statement to create repetition instead of a loop in order to make it easier to establish the connection between `fact` and its flowchart-style FSM representation. In C, `v = e` means that the value of expression `e` will be stored in the memory location named by variable `v`. Such assignments are often denoted as $v \leftarrow e$ or $v := e$ in pseudocode. This must not be confused with a (Boolean-valued) assertion of equality, denoted as `x == y` in C or $x = y$ in mathematics. The notation b `->` α on the FSM means that when (if ever) the Boolean-valued *guard* b is true, then the FSM can execute the statement α in order to perform a state transition.

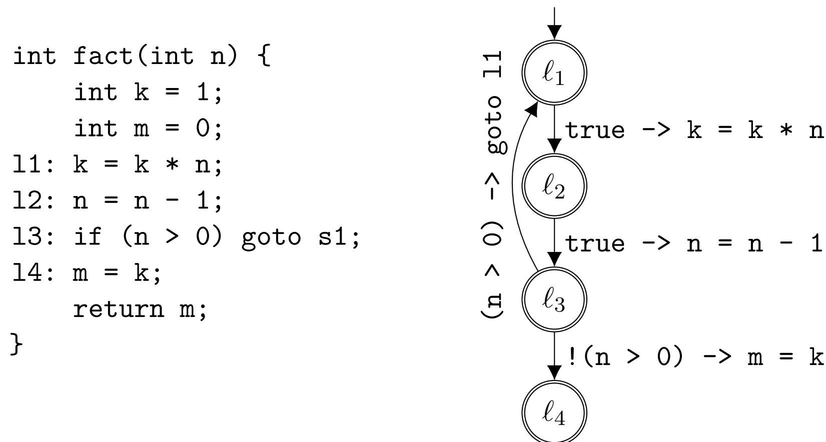

Figure 5: A subroutine in C (on the left) and an FSM (on the right) representing the control logic of the body of that subroutine after the initialization of the variables n, k, and m.

The interpretation of `fact` here as an FSM is intuitive rather than formal. It is somewhat imprecise. Firstly, the complications involved in calling and returning from subroutines are ignored. That is justified in this case, because

this is not an interprocedural analysis [52, Ch. 2.5]. Instead, the FSM represents the logic that takes place inside the subroutine body. Secondly, the FSM only considers the locations of the statements of `fact` and not the values of the variables n, k, and m.

If the variables n, k, and m are taken into account, then the conversion outlined above yields an FSM with $3n + 1$ successive states without loops. Figure 6 demonstrates this when $n = 3$ initially. The reason for why all states are chained one after the other is that `fact` is *deterministic*: At any given state, executing a single statement can only result in a single successor state. Because a state also includes the current location, there can be at most one statement to execute.

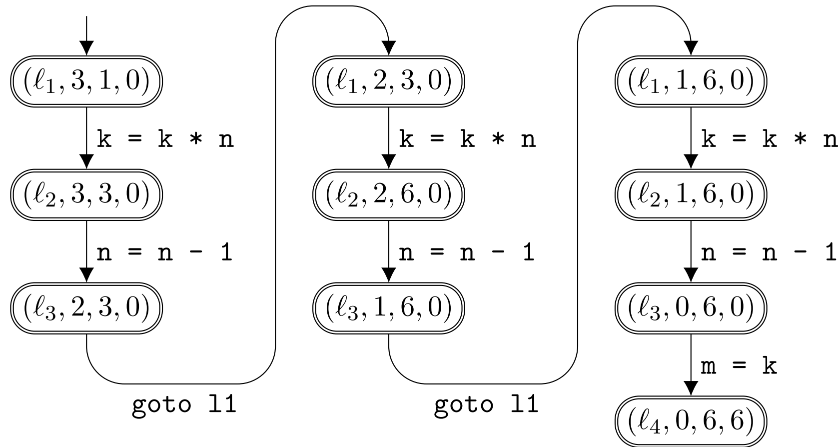

Figure 6: A more accurate FSM representation of `fact` that models states as quadruples (ℓ, n, k, m).

Each choice of n leads into a different FSM, so `fact` describes an infinite family of systems. Model checking that all finite-state models of an infinite family fulfil a property is equivalent to solving the halting problem for an infinite-state system, cf. a Turing machine. The halting problem of Turing machines is undecidable. However, there are also infinite-state systems such as well-structured transition systems [20] that allow decidable state space enumerations. For model checking properties of ADAPOS, the models have been made finite-state by fixing free parameters during compilation.

An *execution* (of an FSM) is an alternating sequence

$$s_0 \alpha_1 s_1 \alpha_2 s_2 \alpha_3 \dots$$

of states $s_i \in S$ and actions $\alpha_i \in \Sigma$ such that $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ for every $i \in \mathbb{N}$. A *finite execution* is a sequence

$$s_0 \alpha_1 s_1 \alpha_2 s_2 \alpha_3 \dots \alpha_n s_n$$

that is an initial segment of an execution.

An FSM *accepts* a finite path $\alpha_1\alpha_2\ldots\alpha_n$ (called *string* in this context) if and only if there is a finite execution $s_0\alpha_1 s_1\alpha_2\ldots\alpha_n s_n$ such that $s_0 = \iota$, the initial state, and $s_n \in F$. A *Büchi automaton* is similar to an FSM, except it accepts an infinite string $\alpha_1\alpha_2\ldots$ if and only if there is an (infinite) execution $s_0\alpha_1 s_1\alpha_2\ldots$ starting from $s_0 = \iota$ that contains infinitely many occurrences of one or more states in F.

The *handshake product* [2, p. 48] $\mathcal{M}_1||\mathcal{M}_2$ of FSMs $\mathcal{M}_1$ and $\mathcal{M}_2$, with $\mathcal{M}_i = (\Sigma_i, S_i, \iota_i, \delta_i, F_i)$ for $i \in \{1,2\}$, is an FSM $(\Sigma, S, s_0, \delta, F)$ such that

- $\Sigma = \Sigma_1 \cup \Sigma_2$;
- $S = S_1 \times S_2$;
- $\iota = (\iota_1, \iota_2)$;
- δ is defined with three rules:

$$\frac{\alpha_1 \notin H, (s_1, \alpha_1, s_1') \in \delta_1}{((s_1, s_2), \alpha_1, (s_1', s_2)) \in \delta}(I_1) \qquad \frac{\alpha_2 \notin H, (s_2, \alpha_2, s_2') \in \delta_2}{((s_1, s_2), \alpha_2, (s_1, s_2')) \in \delta}(I_2)$$

$$\frac{\alpha \in H, (s_1, \alpha, s_1') \in \delta_1, \text{ and } (s_2, \alpha, s_2') \in \delta_2}{((s_1, s_2), \alpha, (s_1', s_2')) \in \delta}(HS)$$

  where $H \subseteq \Sigma_1 \cap \Sigma_2$ is the set of handshake actions; and finally

- $F = F_1 \times F_2$.

The rules I_1 and I_2 express *interleaving* while the rule *HS* expresses *handshaking.* For a non-handshake action, one of the two component FSMs performs a transition while the other component FSM does nothing. If both component FSMs can perform a transition, then the handshake product automaton chooses non-deterministically one of them to perform a transition while the other one does nothing. When the two component FSMs can perform a transition with the same handshake action, then they perform that transition simultaneously. The difference between interleaving and handshaking is analogous to the difference between time sharing and parallel execution.

A railroad crossing system [2, pp. 50 – 52] is an example of a handshake product of three transition systems:

1. A *train*, which is initially far away from the crossing. It is safe to cross the railroad until the train gets near the crossing.
2. A *gate*, which needs to be lowered before the train reaches the crossing in order to prevent accidents. The gate can be opened again after the train has left the crossing.

3. A *controller* that receives information about trains that are about to enter the crossing. The controller must send the signal to raise or lower the gate when appropriate.

A natural correctness property for the railroad crossing system is that whenever the train is in the crossing, the gate must be down.

Figure 7 represents the three component systems as FSMs. Figure 8 shows the handshake product of the train, the gate, and the controller with handshake actions *approach*, *exit*, *lower*, and *raise*. The system a design flaw: The train might enter the crossing while the gate is still up. This unwanted behaviour will be exposed momentarily using an enumerative model checking algorithm.

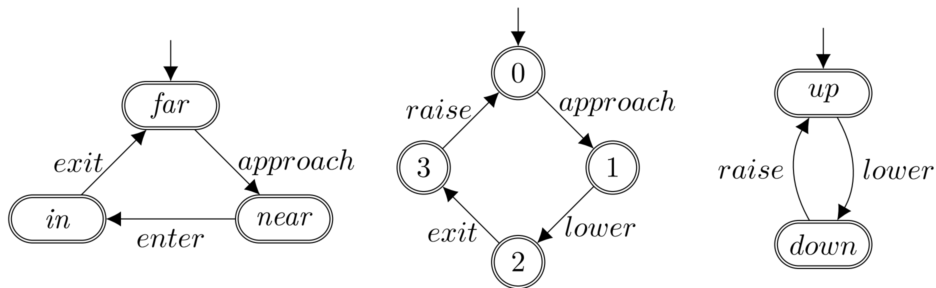

Figure 7: The components FSMs of the railroad crossing system. The components are, from left to right: Train, controller, and gate.

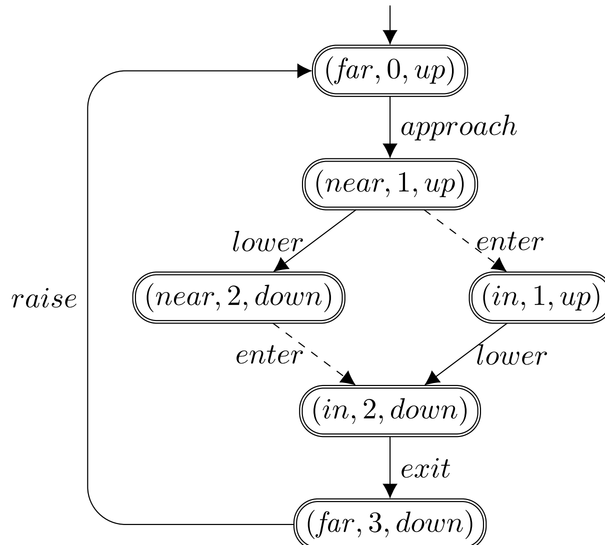

Figure 8: The handshake product of the railroad crossing system. Dashed arrows denote transitions by interleaving (the Int rule) while other arrows denote handshake transitions (by the HS rule).

The *Synchronous product* [61, p. 61] [36, p. 159] [2, pp. 73 – 75] $\mathcal{M}_1 \times \mathcal{M}_2$ of FSMs $\mathcal{M}_1$ and $\mathcal{M}_2$ is a special case of $\mathcal{M}_1 || \mathcal{M}_2$ where $\mathcal{M}_1$ and $\mathcal{M}_2$ share the same alphabet Σ and every transition uses the rule *HS* with $H = \Sigma$. Simply put, $\mathcal{M}_1 \times \mathcal{M}_2$ can only perform a transition when both $\mathcal{M}_1$ and $\mathcal{M}_2$ can. The FSMs $\mathcal{M}_1$ and $\mathcal{M}_2$ in $\mathcal{M}_1 \times \mathcal{M}_2$ perform their transitions simultaneously, or in *lock-step*.

3.2 Linear Temporal Logic

Linear Temporal Logic (LTL) is a formal language used for specifying properties. Though known longer among logicians, LTL was introduced into computer science by Amir Pnueli in 1977 [56]. LTL expresses ordering of events [2, p. 226] (e.g., "the sun rises before it sets"), rather than real-time assertions (e.g., "the monitor shuts down after five minutes of inactivity") or claims about calendar events (e.g., "31 December is the New Year's Eve"). The syntax of LTL formulae over a set A of atomic propositions [2, p. 227] in BNF is:

$$
\begin{array}{lcl}
\langle formula \rangle & ::= & \text{`}\texttt{true}\text{'} \\
 & | & \langle atomic\ proposition \rangle \\
 & | & \text{`('}\ \langle formula \rangle\ \text{`}\wedge\text{'}\ \langle formula \rangle\ \text{`)'} \\
 & | & \text{`}\neg\text{'}\ \langle formula \rangle \\
 & | & \text{`}\bigcirc\text{'}\ \langle formula \rangle \\
 & | & \text{`('}\ \langle formula \rangle\ \text{`U'}\ \langle formula \rangle\ \text{`)'}
\end{array}
$$

In model checking, atomic propositions often establish constraints over values of variables (e.g., $x < 10$) or locations of processes (e.g., "process A is at critical section"). This means that atomic propositions express properties of states. The constant `true` stands for a proposition that holds for every state.

The meta-variables φ and ψ will henceforth refer to arbitrary LTL formulae. Intuitively, $\bigcirc\varphi$ asserts that φ will hold after a single abstract discrete moment of time. (More precisely, $\bigcirc\varphi$ asserts that φ holds for the next state in every possible execution that continues from the current state.) The formulae $\bigcirc\varphi$ and $\neg \bigcirc \neg\varphi$ are equivalent. A formula of the form $\varphi \text{ U } \psi$ asserts that eventually ψ must hold and that φ holds perpetually until ψ becomes true.

Just like in classical propositional logic, the connectives `false`, $\vee$, $\rightarrow$, $\leftrightarrow$ can be derived from $\wedge$ and $\neg$. There are other commonly used LTL operators, such as $\Box$, $\Diamond$, W, and R among others that can be derived from U using propositional logic. The derivations are given below:

$$
\begin{array}{ll}
\texttt{false} \equiv \neg\texttt{true} & \Diamond\varphi \equiv \texttt{true}\ \text{U}\ \varphi \\
\varphi \vee \psi \equiv \neg(\neg\varphi \wedge \neg\psi) & \Box\varphi \equiv \neg\Diamond\neg\varphi \\
\varphi \rightarrow \psi \equiv \neg\varphi \vee \psi & \varphi\ \text{R}\ \psi \equiv \neg(\neg\phi\ \text{U}\ \neg\psi) \\
\varphi \leftrightarrow \psi \equiv (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi) & \varphi\ \text{W}\ \psi \equiv \Box\varphi \vee (\varphi\ \text{U}\ \psi)
\end{array}
$$

Intuitively, $\Box\varphi$ asserts that φ holds always while $\Diamond\varphi$ asserts that φ eventually holds at least once. The combinations $\Box\Diamond\varphi$ and $\Diamond\Box\varphi$ assert that φ holds infinitely often and φ holds eventually forever, respectively.

Figure 9 presents ten examples of LTL formulae along with one of their infinitely many satisfying executions in an unspecified model with respect to arbitrary atomic propositions p, q, and r. Figure 10 presents counter-examples to the same formulae.

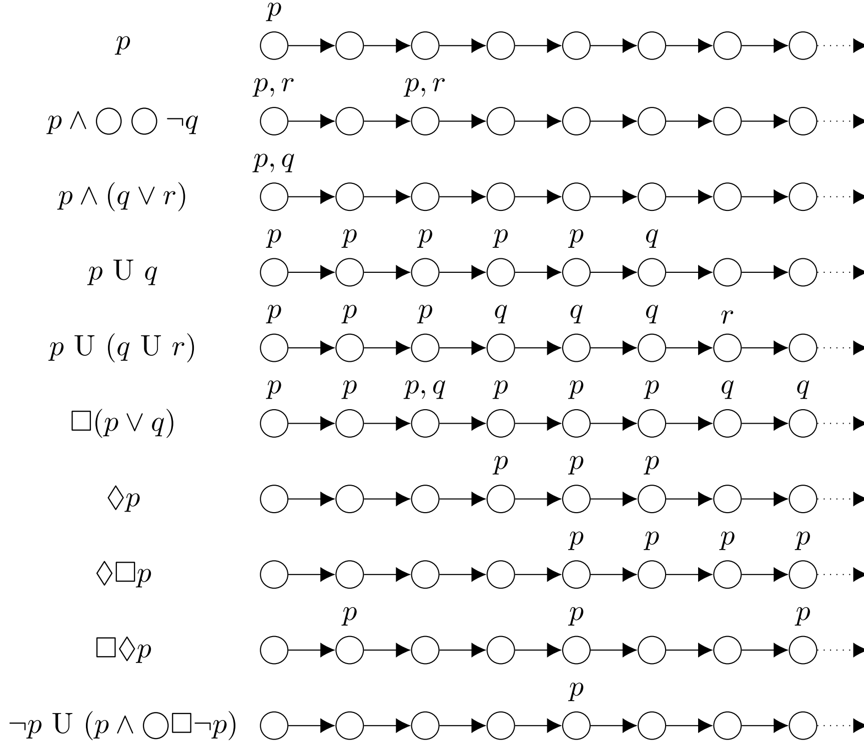

Figure 9: Intuitive semantics of some LTL formulae.

A *Kripke structure* [17, p. 6] (S, R, L) over a set A of atomic propositions has a set of states S, a relation $R \subseteq S \times S$, and a labelling function $L : S \to 2^A$ that assigns each state a set of propositions that the state satisfies. Kripke structures are similar to automata, except that instead of having labels on transitions (i.e. actions), they have labels on states. A *path* on a Kripke structure is a sequence $s_0 s_1 s_2 \ldots$ such that $(s_i, s_{i+1}) \in R$ for every $i \in \mathbb{N}$.

The relation $\pi \models \varphi$ ("π satisfies φ") between (an infinite) path π in a Kripke structure and an LTL formula φ is defined inductively [2, p. 231] [17, p. 8] [42, p. 141]). Denote the path π with the first i states removed as π^i. Let $\pi = s_0 s_1 s_2 ...$ and define

$$
\begin{array}{lll}
\pi \models \texttt{true}; & & \\
\pi \models p & \text{iff} & p \in L(s_0); \\
\pi \models \varphi \wedge \psi & \text{iff} & \pi \models \varphi \text{ and } \pi \models \psi; \\
\pi \models \neg\varphi & \text{iff} & \pi \not\models \varphi; \\
\pi \models \bigcirc\varphi & \text{iff} & \pi^1 \models \varphi; \text{ and} \\
\pi \models \varphi \,\mathrm{U}\, \psi & \text{iff} & \pi^j \models \psi \text{ for some } j \geq 0 \text{ and } \pi^i \models \varphi, \forall i, 0 \leq i < j.
\end{array}
$$

Now enumerative model checking can be defined rigorously.

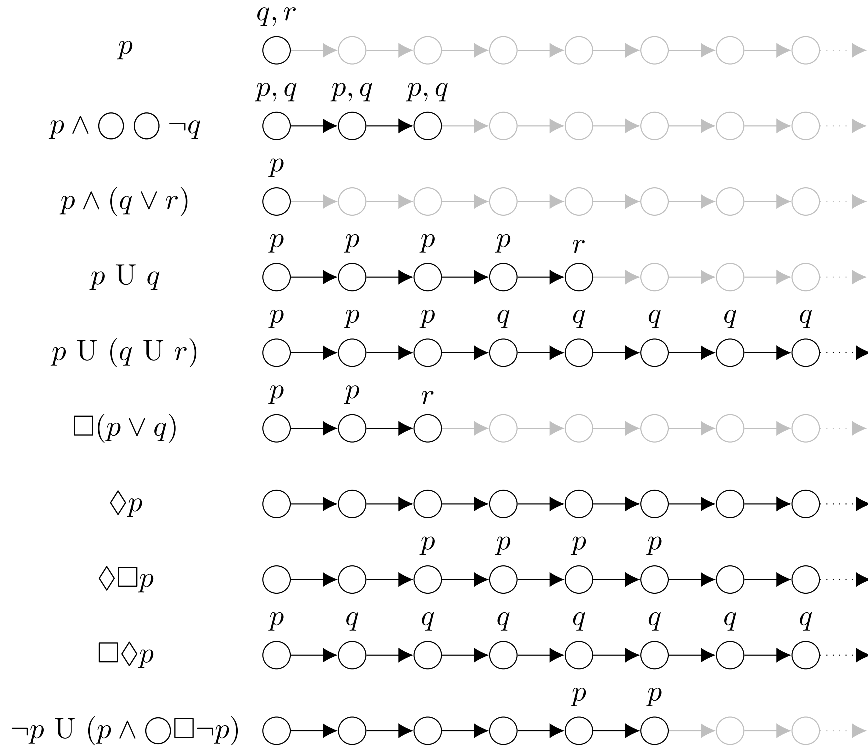

Figure 10: LTL formulae and their corresponding counter-examples. For safety properties, violations can be demonstrated in finitely many steps. The executions drawn in gray continue after the states where property violations have been established.

3.3 Enumerative Model Checking

An LTL formula can be converted into a Büchi automaton algorithmically [26] [42, Ch. 4.6.1]. Gerard Holzmann lists four steps for formalising the model checking problem [36, p. 159]:

1. Convert the negation $\neg\varphi$ of the LTL property φ into the Büchi automaton $\mathcal{B}_{\neg\varphi}$.

2. Compute the handshake product $\mathcal{A} = \mathcal{A}_1 || \mathcal{A}_2 || \ldots || \mathcal{A}_n$[21] of the automata $\mathcal{A}_i$ representing the concurrent processes in the model. $F_i = S_i$ for each component automaton $\mathcal{A}_i$.
3. Compute the synchronous product $\mathcal{C} = \mathcal{A} \times \mathcal{B}_{\neg\varphi}$. Note that $\mathcal{A}$ is an FSM and $\mathcal{B}_{\neg\varphi}$ is a Büchi automaton, but in this case the product can be computed as a synchronous product of FSMs. The resulting automaton will be a Büchi automaton.
4. See if $\mathcal{C}$ has accepting executions.

Because all states in $\mathcal{A}$ are accepting, it is $\mathcal{B}_{\neg\varphi}$ alone that determines whether or not a given execution qualifies as a counter-example. There are depth-first and breadth-first algorithms for enumerative model checking [36].

Consider the railroad crossing system again. A natural safety requirement for the system is that the gate must be always down when the train is approaching, i.e. $\Box$(“train is in” $\rightarrow$ “gate is down”). Figure 11 represents a Büchi automaton that accepts those executions where the train enters the crossing while the gate is still open. Figure 12 shows the synchronous product of the railway crossing system and this Büchi automaton. It has a reachable strongly connected component [42, p. 110] starting from the state $((in, 1, up), C)$ with accepting only states. Any execution visiting this state is a counter-example to the safety requirement.

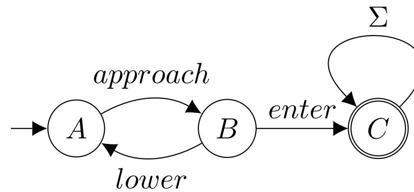

Figure 11: The Büchi automaton accepting executions where the train moves into the crossing while the gate is still up. The label Σ is a wildcard matching all five actions.

There are also other ways of performing model checking. In *symbolic model checking*, the state space of the model is represented in a more abstract way often by grouping states in sets instead of enumerating them individually. Examples of these more abstract state set representation techniques include *Binary Decision Diagrams* (BDD), *Boolean Satisfiability* (SAT) and *Satisfiability Modulo Theories* (SMT). There is a survey focusing on symbolic model checking techniques for LTL properties [58]. Another survey [38] explains many more model checking techniques. Recent books [2, 16] likewise cover many areas of model checking.

[21] Actually, Holzmann uses *interleaving product* which is similar to the handshake product, but without handshake actions H and only the (Int) rule for the transition relation

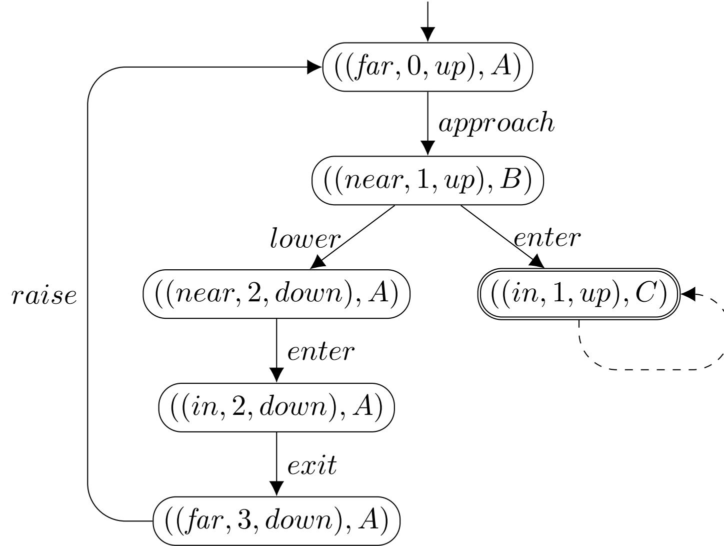

Figure 12: The synchronous product of the FSMs presented in Figures 8 and 11. The dashed arrow means the state $((in, 1, up), C)$ is the initial state of a strongly connected component (shaped like the automaton in Figure 8) made entirely of accepting states.

3.4 State Space Explosion

All of the steps 1 − 4 of the enumerative model checking procedure in Chapter 3.3 can be performed on the fly. If a counter-example exists, then it usually suffices to expand only the part of the state-space covered by that counter-example. However, in worst case, and despite the optimisations mentioned briefly above, enumerative model checking can be computationally extremely expensive. Theorem 2 in [17, p. 13] expresses that:

> "There is an LTL model-checking algorithm whose running time depends linearly on the size of the Kripke structure and exponentially on the length of the LTL formula."

Furthermore, it is known that the enumerative LTL model checking problem is PSPACE-complete [42, p. 147]. As [17, p. 13] and [42, p. 147] point out, in practice the bottleneck is usually the size of the FSM and not the size of the LTL formulae. The extremely fast growth of the number of states of a model as a function of the length of the model's formal description is known as the *state-space explosion* (or just state explosion) problem. There is a survey on state explosion problem [64].

Baier and Katoen [2] Denote Var as the set of all variables of a process, *Loc* as the set of possible program counter values, and $dom(x)$ as the set of values that variable x may take, i.e. the *domain* of x. Thus, the number of

states in the program graph is [2, p. 75]

$$|S| = |Loc| \cdot \prod_{x \in Var} |dom(x)|, \tag{1}$$

where $|X|$ denotes the cardinality (i.e. the number of elements) of the set X. Under the translation outlined in Chapter 3.1, the Equation (1) gives the number of states of the FSM representation of the process.

Without loss of generality, the domain of program counter values and the domains of variables may be assumed to be sets of bit strings. Thus, the equation (1) may be estimated as

$$|S| \leq 2^k \cdot \left(\prod_{0<i\leq|Var|} 2^m \right), \tag{2}$$

where k is the number of bits needed for representing the domain of program counter values and m is the number of bits needed for representing the variable with the largest domain. If there are n variables, then (2) becomes

$$|S| \leq 2^{kmn}. \tag{3}$$

If there are p such processes, then the upper bound becomes

$$|S| \leq 2^{pkmn}. \tag{4}$$

For a system with heterogeneous processes, the upper bound can be estimated in terms of the process with largest state space and assuming that all processes are copies of the largest process. Thus, the state space potentially grows exponentially in at least four different variables.

Not all states in a model are necessarily reachable, though. In general, establishing the number of reachable states is a much more difficult task than estimating the upper bound for the number of all states. Determining the reachable states requires either deep knowledge in the details of the particular model under investigation or a large amount of computation for enumerating the reachable portion of the state space.

Consider the following C code fragment.

```
int x = 0; int y = 0;   // Initial state
if (1) {                // 1 means true in C
    t0: x = 1; goto t1; // Set x to 1, go to next transition
    t1: x = 0; goto t0; // Set x to 0, go to other transition
} else {
    t2: y = 1;          // Unreachable
}
```

On an abstract level, the fragment above could be modelled as a (deterministic) finite state automaton or as a Kripke structure, as shown in Figure 13. The transitions of the finite state machine are labelled according to the labelling in the C code, while the states of the Kripke structure are labelled with the values of the variables x and y. The state s_0 is the state after the initialization of the variables x and y in the first line of the C code fragment. s_1 is the state after transition t_0 and before the transition t_1, which takes the process back in state s_0.

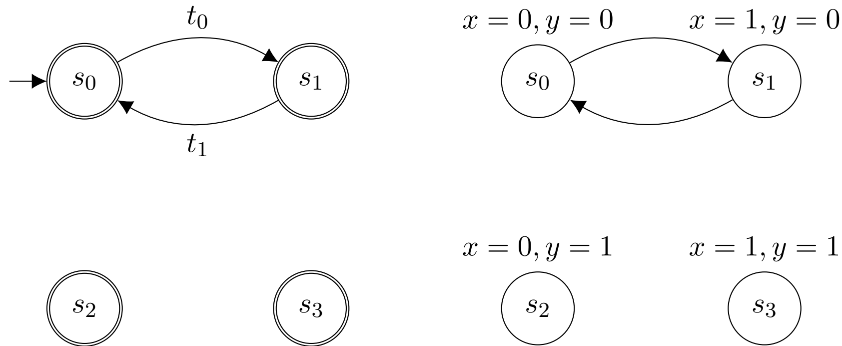

Figure 13: A finite state machine (on the left) and a Kripke structure (on the right) representing the C code fragment.

Notice that the transition t_2 from s_0 to s_2 has been omitted from the diagrams, because it is never *enabled*: The condition of the `if-else` control structure can never be false, so the line labelled with `t2` is *dead code*. Also, there is no code at all which could move the process to or from the state s_3. Hence, the states s_2 and s_3 are *unreachable*. Therefore, the size of the reachable state space for the Kripke structure is two, even though the whole state space has size four, assuming that the variables x and y are stored as single bits.

More realistically, all variables of type *int* might take one or more words of memory, e.g., 32 or 64 bits. If x and y would be stored as 32-bit integers, then there would be 2^{32+32} potential combinations of their values. Furthermore, even though the code conceptually has only three transitions, the actual number of control points in the C code is eight. This makes the total number of states $8 * 2^{64}$ as per equation (1). Fortunately, enumerative model checkers only allocate memory for the states they can reach, so the states for C code example above would be easy to enumerate.

The C code example also shows that a program having finitely many states does not need to terminate. This makes model checking suitable for verifying properties of *reactive systems*. Reactive systems in real life include control systems (including the ALICE [14] and CMS [37] DCSs), operating systems, and server processes. In fact, for many reactive systems, including ADAPOS, termination considered as a violation. By contrast,

many *deductive verification* [57] techniques that rely on theorem proving hinge on proofs of termination.

Another source for state space explosion is branching. The following complete PROMELA model is an extreme example of branching:

```
init {
  byte x; byte i;
  for (i : 0 .. 7) {
    if
      :: x = x | (1 << i)    /* Maybe sets the ith bit... */
      :: skip                /* or maybe does not         */
    fi
  }
}
```

This PROMELA model essentially represents a random number generator that produces a random byte by eight consecutive coin flips. The bytes x and i are implicitly initialized to zeros. Inside the deterministic `for`-loop, there is a non-deterministic `if`-statement with two `then`-branches which either sets the ith bit of the byte x to 1 or leaves x unchanged. Both cases will be investigated by the model checker SPIN. After the loop has been executed, x may have any value, depending on the choices made during the loop. According to the verifier executable generated by the SPIN model checker, this model has 1024 reachable states. Figure 14 illustrates the shape of this model's state space.

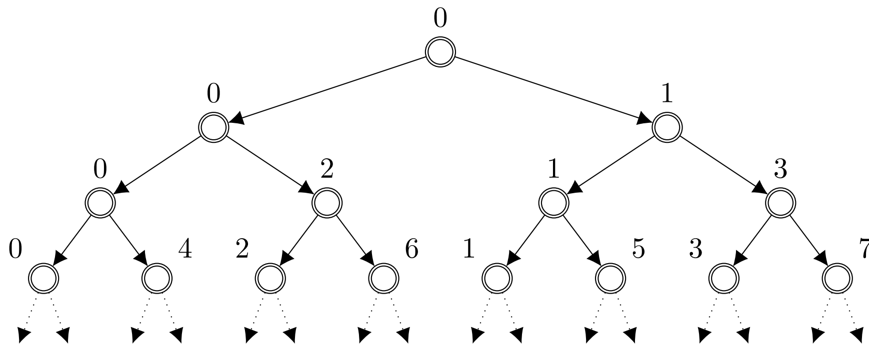

Figure 14: The state space of the PROMELA model unrolled for the first three steps of the `for`-loop. The states are labelled according to the value of the variable x (omitting the '$x =$' part of the propositions). Names for the states have been omitted, too.

To summarize the state-space explosion problem from a practical perspective, a model for a reactive system needs to try to minimize

- the number of processes;
- the number of statements;

- the amount of non-determinism;
- the number of variables;
- the size of the domains of the variables;
- the number and size of arrays and channels;
- the number of possible interleavings of statements belonging to concurrent processes;
- the number of auxiliary variables used for formulating the properties; and
- lengths of the LTL formulae and the sizes of their corresponding Büchi automata.

The model should not have any unnecessary sources for complexity, because model size tends to be the bottleneck of model checking. Even though the number of reachable states is often many orders of magnitude lower than the number of all states, it is still often a very large quantity.

In the previous study [49], two model checkers were used for verifying properties of ADAPRO on both design and implementation level. The plan was originally to follow a similar approach in the current study. Implementation level C++ models were drafted for Engines and Terminals as individual processes with a simulated non-deterministic environment. These models were meant to be checked using the model checker DIVINE [4].

The complexity of the modelling task necessitated narrowing the focus to design level only, to make the work fit a master's thesis. Based on earlier good experiences [49], the model checker SPIN [34] was chosen for building and verifying the model of ADAPRO discussed in this thesis.

4 The Model of ADAPOS

Now that the necessary background information for ADAPOS has been discussed, the following chapters will dive in to the technical substance. ADAPOS is a tailor-made system designed to serve a specific purpose on a specific environment operated by specialists over the limited duration of the third run of LHC. The model will be explained operationally on a moderately high level of detail. The general idea of the model will be presented first. After that, the eight variants refining certain details of the general specification will be introduced.

An abstract model of asynchronously operating ADAPOS Engines and Terminals was built with PROMELA [22] for two reasons. Firstly, the model serves as an abstract specification document, focusing on the essential structures and phenomena and leaving out irrelevant details. Building a model is a great way to learn new insights about the domain of an application. Secondly, the C++ implementation-level models for Engine and Terminal that were initially planned had huge state spaces. Composing the C++ models together and verifying them would have been technically and computationally quite a challenge. The C++ models had to be dropped eventually.

The PROMELA model is summarised in Figure 15. There are three types of concurrently operating processes, a Load Generator (LG), Engines, and Terminals. All processes loosely follow FSM-like logic and have their own copies of the FBI, represented as vertically aligned arrays in Figure 15. As Figure 15 suggests, all FBIs have exactly the same size. Data points are represented as bytes. For data connections, regular arrows denote single data point transmissions while bold arrows denote single data point or FBI transmissions depending on Engine state. Dotted arrows denote control connections for requesting FBI transmissions.

The model has four compile-time constants:

- *Service count* determines the number of data points per FBI.
- *Engine count* determines the number of Engines.
- *Terminal count* determines the number of Terminals.
- *Event count* determines how many updates to data points LG produces.

The states of FSMs shown in Figure 15 are high-level abstractions like the FSMs shown in Figures 5, 7, and 8 in Chapter 3.1, and also Figures 11, 12, and 13 in Chapter 3.4. The states of the FSMs of Engines and Terminals in Figure 15 are called *protocol states*. Protocol states should not not be confused with ADAPRO states which are omitted from this model.[23]

[22] Version 6.10 of the model is archived at https://doi.org/10.5281/zenodo.4767686

[23] See Section 3.2 in the manual of ADAPRO.

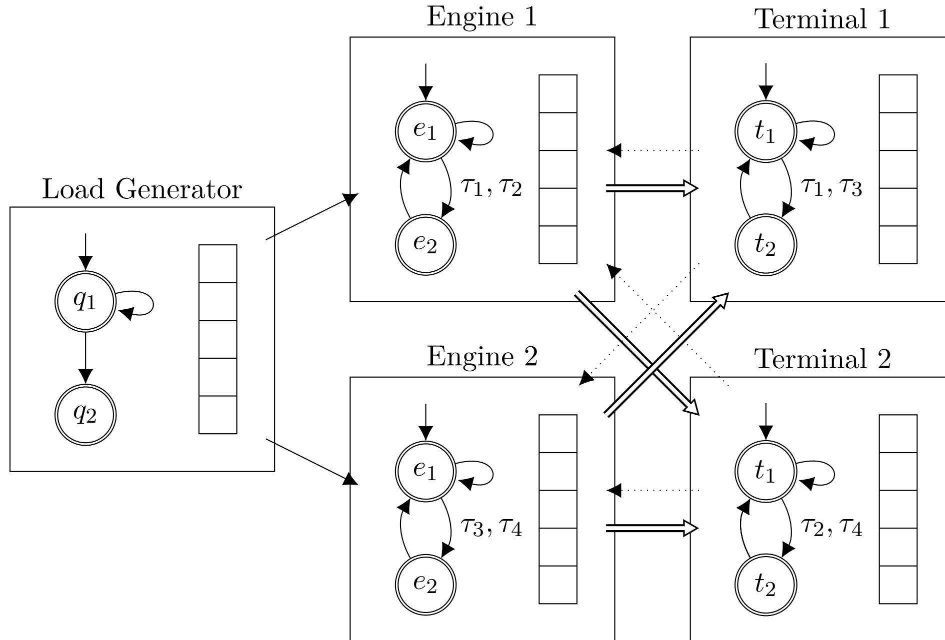

Figure 15: A schematic view on the abstract model of ADAPOS with five data points per FBI, a Load Generator, two Engines, and two Terminals. The handshake actions τ_i are used for synchronisation.

Protocol states could be considered to be substates of the ADAPRO state `RUNNING`, though this is a slightly inaccurate simplification.

The model makes the following assumptions on publish-subscribe network connections with DIM [25] and 0MQ[24,25] protocols:

- Message contents are never altered.
- Messages between two processes are not duplicated in transit.
- Those messages between two processes that are received arrive in the order they were sent.
- When there is a message in transit between two processes and the sender attempts to transmit another message, then one of these messages will be dropped. DIM drops the old message and 0MQ drops the new one.
- Messages to n recipients are transmitted over n one-to-one connections which may drop messages independently.
- Engines may disconnect from Terminals for unspecified reasons.

[24] TCP is assumed to be the underlying transport protocol.
[25] https://rfc.zeromq.org/spec/29/

The data connections from LG to Engines and Engines to Terminals interpret the publish-subscribe patterns of DIM and 0MQ respectively. The control connections from Terminals to Engines represent DIM command calls that are assumed to never fail or be delayed.

The large state space of the base PROMELA model and the fact that there are multiple interpretations on how DIM and 0MQ connections could be modelled led to the creation of several variants of the model. Variants are maintained in separate branches in the Git version control system used in the modelling project. The variants differ by their interpretations of *input* (LG to Engines) and/or *output* (Engines to Terminals) connections.

Figure 16 illustrates refinement relations between the variants of the model. In object-oriented terms, the arrows point from inheriting variants to the more general ones. This hierarchy is only conceptual and it is not reflected in the PROMELA code in any way.

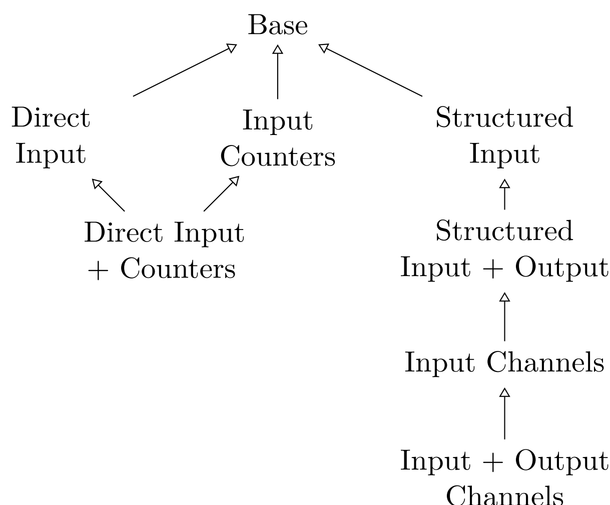

Figure 16: The variants of the PROMELA model of ADAPOS.

Chapters 4.3 – 4.10 sketch a simplified overview on these variants using Python-style pseudocode instead of PROMELA. A book [6] provides a hands-on introduction to the PROMELA modelling language and the SPIN model checker. For the sake of conciseness, certain details on the variants have been omitted from this presentation, including

- details of the FSMs shown in Figure 15;
- FBI transfers between Engines and Terminals; and
- details on how Terminals keep track on the freshness, the current value, and the previous value of a data point in their FBIs.

General discussion on the roles and logic of the different process types of the model is appropriate before introducing the variants, though.

4.1 Processes in the Model

LG and Engines are agnostic about other processes receiving their outputs. They act as publishers sending the same data to any number of subscribers. It is possible that a message will be received by some subscribers while it is dropped for others. Neither of these two process types ever block when sending messages. Both process types follow the same general logic regardless of the number of connected subscribers.

LG simulates the DIM services producing input to ADAPOS. By default, LG generates distinct updates until the number of updates generated exceed the event count. At this point, LG halts. LG also has an alternative cyclic mode in which it never halts, and keeps cycling through three constant values per data point. The event count parameter has no effect when LG is compiled to run in cyclic mode.

Consider the FSMs of Engines and Terminals shown in Figure 15. In the PROMELA code, the states e_1 and t_1 are both named as `SEND_SINGLE`. The states e_2 and t_2 share the name `SEND_FBI`. The meaning of these protocol states is that during e_1 and t_1 the processes expect the messages to be single data points, while state e_2 and t_2 the processes expect the next message to be an FBI.

As a simplification, the states e_1 and t_1 correspond approximately to the loop in Figure 2 in Chapter 2.2. The states e_2 and t_2 correspond to the exchange of messages between "DIM(sendMaster)" and "copy(FBI', FBI)". Only the FBI transmission using 0MQ ("0MQ(FBI')") and replacing the Terminal's FBI with the FBI received from Engine ("copy(FBI', FBI)") are included in the model.

During the state e_1, an Engine updates data points in its FBI whenever it receives updates to them from LG. After updating its FBI, the Engine sends the updated data point to Terminals. Engine moves to the state e_2 only when a Terminal requests it to do so. During the state e_2 the Engine transmits its FBI to Terminals. After dispatching its FBI, Engine returns to state e_1.

A Terminal begins by establishing a connection to an Engine in state t_1. When connecting to an Engine, the Terminal tells the Engine to move to state e_2 as it transitions to state t_2 itself. This can be thought of as a handshake action, denoted as τ_i for $i \in \{1, 2, 3, 4\}$ in Figure 15. For example, when Terminal 1 connects to Engine 2, they both perform the transition associated with τ_3.

During the state t_2, the Terminal is expecting to receive an FBI from the Engine. If it receives a single data point, it goes back to state t_1 and establishes a connection to another Engine. If it receives an FBI, it copies its contents into its own FBI and moves into state t_1. After that, Terminal stays in t_1 copying both single data points and FBIs into its FBI as they arrive. A disconnection always makes the Terminal to switch to the next

Engine and move back to state t_1.

Engines follow their two-state FSM logic accurately. For Terminals, the FSM is more of an abstraction. Terminals need to (re)establish connections with Engines, which is not reflected by their FSMs. Perhaps Terminals could have a third state in their FSMs for (re)connecting to Engines. However, an Engine does not need to know when a Terminal is considering to switch to another Engine. It is simpler to think in terms of two protocol states for Engines and Terminals.

Ideally, Engines and Terminals should always change their states in lockstep. Initially, a pair of an Engine and a Terminal starts from the states e_1 and t_1 respectively. Next, they should both agree to transfer the FBI, moving to states e_2 and t_2 respectively. After that, they should stay in states e_1 and t_1 respectively forever. The trace of this ideal execution would be $(e_1, t_1)(e_2, t_2)(e_1, t_1)(e_1, t_1)(e_1, t_1)\ldots$ for each Engine-Terminal pair.

4.2 Differences Between the Model and the Implementation

This chapter discusses some finer semantic details of the model. It can be skipped on the first reading.

An Engine may non-deterministically disconnect from any Terminal at any point. It is also possible that the Engine disconnects from all Terminals at the same time. This could be interpreted such that the Engine shut down unexpectedly or got disconnected from the network.

If an Engine disconnects from one or more Terminals, the Engine will continue operating normally. The shutdown and (re)start sequences of Engines are omitted from the model. The argument for this decision is that properties of the model focus mainly on Terminals. Terminals should not be sensitive to the internal states of Engines and neither should the model be. More importantly, the DAQ readout process receiving data from ADAPOS should not even need to be aware of Engines at all.

From the point of view of Terminals, it is not relevant what was the root cause for a disconnection. Whenever an Engine disconnects from a Terminal, the Terminal will establish a connection with another Engine. For limiting the size of the state space, Terminals always manage to connect to Engines without errors. Nothing prevents an Engine immediately disconnecting after a connection has been established, so this simplification should not hide errors in the design.

For a 0MQ publish-subscribe connection, a parameter known as *high water mark*[26] (HWM) determines how many messages between a publisher and a subscriber there can be in transit. Once the HWM is exceeded, the publisher starts dropping messages until there is room for new messages once more. The messages that get through are those that were sent before the

[26] https://zeromq.org/socket-api/#high-water-mark

HWM was exceeded (though for ADAPOS, it would be preferable to start dropping messages starting from the oldest one).

The PROMELA model behaves approximately as if HWM was just 1. This design choice makes the model rather pessimistic, as it makes it easy to find executions during which messages are dropped. On the other hand, the model may hide some behaviours that could result from buffering when HWM was greater than 1. As buffering is essentially just copying values without modification, it is not an interesting detail from the safety perspective. This buffering would would further aggravate the state explosion problem, which explains the current design choice to elide buffering in network connections.

Thanks to multi-threading, real ADAPOS processes are able to simultaneously receive and send data over the network. In the PROMELA model, an Engine may either update its FBI or broadcast it to Terminals, but not both at the same time. If Terminals keep constantly bombarding the Engine with requests for FBIs, the Engine will not have a chance to update the contents of its FBI. Thus, Terminals will receive stale data, resulting in a *denial of service* situation.

The unwanted behaviour described above would not be possible in the real system. Such unrealistic behaviours arising from simplifications made in the modelling or model checking process are called *spurious*. Distinguishing spurious counter-examples from realistic counter-examples is one of the challenges in verifying liveness properties. The existence of this particular problematic behaviour could not be confirmed with SPIN though the model checker did find other kinds of liveness violations. Liveness properties had to be ultimately left out from the scope of this thesis.

Terminals only request one FBI per connection. A flood of FBI requests would likely occur during a malicious attack. In production environment, ADAPOS machines will not be directly exposed to the Internet. Thus, malicious attacks are expected to be much more unlikely to happen than honest bugs to surface. This thesis does not investigate data security, but the ARP case study [9], for example, shows that model checking can be also used for finding security exploits.

The PROMELA model has other significant differences from the real ADAPOS software. The main reason for the differences is the need for keeping the state space as small as possible. Minimizing state space size necessitates leaving details out from the model. Notable differences, which the C++ models were meant to investigate in more detail, are listed below:

- The most obvious difference between the PROMELA model and real ADAPOS is that there is no upper bound for the number of updates in the real system. ADAPOS could run for an arbitrarily long period of time. The amount of memory needed by ADAPOS should remain almost unchanged after start-up. This could be verified in the C++ models. Hence, ADAPOS has finitely many states in theory, but the

actual state space is probably unimaginably large. The event count parameter is important for adjusting the size of the model.

- All integer variables are in the Promela model are unsigned bytes. All values in the model are bounded above by compile-time constants. There is also a hard limit of 255 processes in Promela.

- The model represents FBIs as small byte arrays, not as huge DPCOM vectors. The reason is simply that full DPCOM structure is not needed for verifying the properties of interest. This choice of representation implies that all accesses to data points in the model are atomic, making property 1 in Chapter 2.3 trivially true. The C++ implementations of ADAPOS processes use locking to ensure mutually excluded DPCOM access.

- In the model, all FBIs and memory allocations in general, have fixed size determined during compilation time. Real ADAPOS Engines determine and fix their FBI sizes during startup. In the production environment, FBI sizes may only change when the configuration of data points in ALICE DCS changes. The configuration of the DIM services providing data points to ADAPOS may only change during LHC technical stops. It is acceptable to require a full reconfiguration and restart of ADAPOS when new data points need to be added to the system.

- The model forgets about the ADAPRO FSM aspect of Engines and Terminals. ADAPRO FSM mechanism has already been studied in a general setting [49]. Omitting the intricacies of ADAPRO makes the model much simpler and more importantly, smaller. The logic represented in the Promela model mostly takes place when the Engine and Terminal processes are in ADAPRO state `RUNNING`.[27]

- Keepalive messages have been omitted from the Promela model. Their only purpose is to detect if the connection between an Engine and Terminal fails. The keepalive mechanism has been simply replaced with the event where an Engine disconnects.

- Another omitted message type are the so-called master DPCOMs. Unlike Engines, Terminals do not know in advance what services ADAPOS publishes. When an Engine is about to broadcast its FBI, it first broadcasts a special master DPCOM which contains the number of DPCOMs in the FBI. This way, Terminals can (re)allocate the right amount of memory for their FBIs. Figure 2 shows the transmission of a master DPCOM over 0MQ as the message “0MQ(master(size’))”.

[27] The only exception is that Terminal establishes its first connection to an Engine during the state `STARTING`. Even this behaviour could easily be moved to the state `RUNNING`.

- When connecting to an Engine, a Terminal clears the messages in transit between the Engine and the Terminal. According to the specification of the 0MQ publish-subscribe pattern, a publisher socket "SHALL create a queue when a subscriber connects to it. If this subscriber disconnects, the [publisher] socket SHALL destroy its queue and SHALL discard any messages it contains."[28] An Engine keeps relaying updates as it receives them, without any knowledge about Terminals. It is thus simpler to make the Terminal responsible for clearing the old messages.

Compared to the PROMELA model of ADAPRO [49], the PROMELA model of ADAPOS has more coarse level of atomicity. The model of ADAPOS uses `atomic` and `d_step` blocks extensively to keep the state space size as small as possible. Aside from the state space size considerations, using coarse atomics is also semantically meaningful. Engines and Terminals are separate processes running on different machines. Their behaviour should not be sensitive to the internal states and transitions of remote processes. Only the messages passed from a process to another should have effect on the high-level behaviour of the system.

4.3 The Base Variant

The *base variant* of the PROMELA model represents network connections through shared memory. An array of data points (i.e. bytes), called the *input array*, in this variant, is exposed to Engines as a global array. The bold arrows in Figure 15 are represented through another global array, called the *output array*.

In what follows, an *atomic* sequence $\alpha_1\alpha_2\ldots\alpha_n$ of statements can be interleaved with another statement or sequence β only as either $\alpha_1\alpha_2\ldots\alpha_n\beta$ or $\beta\alpha_1\alpha_2\ldots\alpha_n$ (and not e.g., $\alpha_1\beta\alpha_2\ldots\alpha_n$). There are two exceptions to this restriction:

- If a process blocks during an atomic sequence, then other processes can be interleaved. (The situation where all processes block is called a *deadlock* and it is considered as a basic liveness violation.) When blocking ends, the process continues execution as if it entered a new atomic block.

- If a process halts or jumps out of an atomic sequence, then it relinquishes the right for non-interleaved execution with other processes.

This notion of atomicity is similar to how locks work in many programming languages. Depending on the particular sequence, atomicity is represented using either `atomic` or `d_step` constructs. Atomic pseudocode sequences using either of the two exceptions above are represented with `atomic` blocks

[28] https://rfc.zeromq.org/spec/29/

in PROMELA while the rest of the atomic sequences in pseudocode are represented with more restricted `d_step` blocks [6, Ch. 4.4].[29]

The first step at explaining the model is to define the basic domains in which contain the values of the model. The basic datatype for most variables is the unsigned byte type, that is, the set $\{0, 1, \ldots, 255\}$. Let

- $X \subseteq \{0, 1, \ldots, 255\}$ be the set of possible data point values (bytes);
- $M = \{\texttt{NONE}, \texttt{DC}\}$ be the set of special constants (with $X \cap M = \emptyset$);
- $c, s, e, t \in \{1, 2, \ldots, 255\}$ be the event count, service count, Engine count, and Terminal count respectively;
- D be the set $\{0, 1, \ldots, s-1\}$ of data point indices;
- E be the set $\{0, 1, \ldots, e-1\}$ of Engine indices;
- T be the set $\{0, 1, \ldots, t-1\}$ of Terminal indices;
- $T' = \{j + e \mid j \in T\}$ be the index set T offset by the number of Engines;
- $In : D \to X$ be the input array;
- $Out : E \times T \times D \to X \cup M$ be the output array; and
- $FBI : (E \cup T') \times D \to X$ be the FBIs of Engines and terminals.

All the sets above are assumed to be finite and non-empty. Variables ranging over the sets D, E, T, and X will be denoted with k, i, j, and v respectively, following the variable naming convention of the PROMELA code. The symbols occurring above will be used in the following Chapters with the same meaning unless otherwise stated or implied by the context. The initial values for the arrays will be

$$\begin{aligned} In[k] &= k+1, && \text{for all } k \in D; \\ Out[i,j,k] &= \texttt{NONE}, && \text{for all } i \in E,\ j \in T,\ k \in D; \text{ and} \\ FBI[x,k] &= 0, && \text{for all } x \in E \cup T' \text{ and } k \in D. \end{aligned}$$

In the base variant, there is little coordination between LG and Engines. The Engines do not know when LG updates the input array. LG uses the local variables k, z, and a global variable end which is initialised to `false`. Algorithm 1 shows essentially the full logic of LG in pseudocode. Lines 2 – 6 correspond with the conceptual state q_1 in Figure 15 while the last line can be thought of as the state q_2. $a \,\%\, b$ denotes the remainder of a/b.

Algorithm 1 Base variant Load Generator in default mode.

```
1: process LoadGenerator
2:     local variable k ← 0
3:     for z from 0 to c − 1 do
4:         atomic
5:             In[k] ← In[k] + s
6:             k ← (k + 1) % s
7:     end ← true
```

[29] Also in the situations where execution can jump into an atomic block, `atomic` is used instead of `d_step`.

Statements from other processes cannot be executed while LG is executing the statements in lines 5 and 6, because they are part of an atomic sequence starting at line 4. However, the loop starting from line 3 is not part of an atomic sequence statements, so other processes may be executed between iterations of the loop. In the cyclic mode, z is restricted to the values $\{0, 1, 2\}$ and LG runs the modified procedure shown in Algorithm 2.

Algorithm 2 Base variant Load Generator in cyclic mode.

```
1: process CyclicLoadGenerator
2:     local variable k ← 0
3:     local variable z ← 0
4:     repeat
5:         atomic
6:             In[k] ← zs + k + 1
7:             k ← (k + 1) % s
8:             if k = 0 then
9:                 z ← (z + 1) % 3
```

There are two differences between the default mode and the cyclic mode of LG. The first one is that z is cycled by incrementing it by one, modulo three, every time k reaches zero. This is similar to how the minute hand on a watch takes one step every time the second hand returns to zero. The other difference is that instead of sending $In[k] + s + 1$, i.e. the previously sent value incremented by s, to the Engines, LG sends $zs + k + 1$. It is straightforward to adapt these two changes between the two operating modes of LG to other variants, too.

The properties to be discussed in Chapter 5 would allow LG to choose k non-deterministically from D. Non-deterministic scheduling of updates, that preserves the order of updates to individual data points in a *sequentially consistent* [45, p. 690] manner would reflect the expected operating conditions of ADAPOS more realistically. Non-determinism is a known source of state space explosion. This particular case of non-determinism was not considered to be essential, so it was omitted from the model at the risk of losing potential counterexamples.

LG cycles over the data point indices D in both operation modes, but in cyclic mode, it also cycles over their content X. When $s = 3$, we can think of In as a triple $(x_0, x_1, x_2) \in X \times X \times X$. Thus, the initial state of In can be expressed as $(1, 2, 3)$. Table 1 illustrates the evolution of k, z, and In after each iteration of the cyclic LG procedure.

In state e_1, Engine number i keeps polling over the input array as shown in Algorithm 3. The procedure MULTICAST called inside the **atomic** block starting from line 12 is part of the same atomic sequence. PROMELA does not support procedures, so the contents of MULTICAST are inlined in PROMELA.

On lines 3 – 8 Engine chooses non-deterministically between sending

k	z	In	Comment
0	0	$(1, 2, 3)$	Initial state.
1	0	$(1, 2, 3)$	$zs + k + 1$ evaluated to 1. Wrote that into $In[0]$.
2	0	$(1, 2, 3)$	$zs + k + 1$ evaluated to 2. Wrote that into $In[1]$.
0	1	$(1, 2, 3)$	Since k reached 0, z was incremented by 1 (mod 3).
1	1	$(4, 2, 3)$	
2	1	$(4, 5, 3)$	
0	2	$(4, 5, 6)$	Since k reached 0, z was incremented by 1 (mod 3).
1	2	$(7, 5, 6)$	
2	2	$(7, 8, 6)$	
0	0	$(7, 8, 9)$	Now k and z have their initial values again.
1	0	$(1, 8, 9)$	
2	0	$(1, 2, 9)$	
0	1	$(1, 2, 3)$	The loop starts again.

Table 1: An execution of LG. It runs into a loop that starts from the fourth state.

$FBI[i, k]$ or `DC` to Terminal j. Informally, this means that the connection between Engine i and Terminal j may or may not fail during any transmission. (ADAPOS must survive regardless.) The verifier generated by SPIN will explore both outcomes which creates two branches in the state space.

The choice between successfully sending the data point or disconnecting is the only source of explicit non-determinism in the model. The condition on line 5 stays false if Terminal j has not yet read its incoming data point before Engine i continues executing. Terminal j loses an update in this scenario. Losing a data point is the third possible outcome after a successful transfer and a disconnect. It is the result of the implicit non-determinism of process scheduling.

Engine updates at most one data point in its FBI during the atomic sequence. If Engine does not disconnect, it reaches line 6. In the condition of the **if** statement on that line, it checks if there is room for a message to be sent over the network to Terminal j. If there is, Engine sends $FBI[i, k]$. If not, the Engine drops the message and carries on. The base variant of the PROMELA model assumes that there is capacity for exactly one data point per triple (i, j, k) in Out.

Terminals can detect incoming data points from Engines through the use of the special constant `NONE`. When Terminal j connected to Engine i attempts to read from $Out[i, j, k]$, it follows a procedure that is given in a simplified form in Algorithm 4. The PROMELA code corresponding to the pseudocode statement **block until** blocks the execution of Terminal j at least until the expression $Out[i, j, k] \neq$ `NONE` becomes true (if it ever becomes). When blocking ends, Terminal continues its execution atomically.

Algorithm 3 Simplified overview of base variant Engine i, restricted to the actions during the state e_1.

```
1:  procedure MULTICAST(i, k, v)
2:      for j in 0 to t − 1 do
3:          flip coin
4:          if heads then
5:              if Out[i, j, k] = NONE then
6:                  Out[i, j, k] ← v
7:          else
8:              Out[i, j, k] ← DC

9:  process ENGINE(i)
10:     local variable k ← 0
11:     repeat
12:         atomic
13:             if end then halt
14:             if In[k] > FBI[i, k] then
15:                 FBI[i, k] ← In[k]
16:                 MULTICAST(i, k, FBI[i, k])
17:             k ← (k + 1) % s
```

Algorithm 4 Simplified overview of base variant Terminal j, restricted to state t_1.

```
1:  process TERMINAL(j)
2:      local variable i ← 0
3:      start: CONNECT(i)
4:      repeat
5:          atomic
6:              block until Out[i, j, k] ≠ NONE
7:              if Out[i, j, k] ≠ DC then
8:                  FBI[j + e, k] ← Out[i, j, k]
9:                  Out[i, j, k] ← NONE
10:             else
11:                 i ← (i + 1) % e
12:                 goto start
13:             k ← (k + 1) % s
```

As explained in Chapter 4, the procedure CONNECT called on line 3 establishes a connection between Engine i and Terminal j using something like a handshake action, setting their states to e_2 and t_2 respectively. After that, the FBI will be transferred. As said above, FBI transfers have been omitted from this presentation. The pseudocode shows what happens after Terminal moves back into state t_1.

This is a straightforward interpretation of the semantics of ADAPOS, but it has disadvantages. While LG (in default mode) and Terminals always do productive work, Engines may have executions in which they keep polling their input arrays causing LG and Terminals to starve. Table 2 shows one such execution. Executions where Engines cause other processes to starve are unfair in the sense that they violate *weak fairness* [2, p. 128]. Unfair executions can make some desirable properties generally invalid for a model.

k	In	$FBI[i]$	Comment
0	$(1,2,3)$	(0,0,0)	Initial state.
1	$(1,2,3)$	(1,0,0)	$In[0]$ was greater than $FBI[i][0]$.
2	$(1,2,3)$	(1,2,0)	$In[1]$ was greater than $FBI[i][1]$.
0	$(1,2,3)$	(1,2,3)	$In[2]$ was greater than $FBI[i][2]$.
1	$(1,2,3)$	(1,2,3)	No change in $In[0]$. Loop starts.
2	$(1,2,3)$	(1,2,3)	No change in $In[1]$.
0	$(1,2,3)$	(1,2,3)	No change in $In[2]$.
1	$(1,2,3)$	(1,2,3)	The loop starts again.

Table 2: A looping execution of Engine i that makes other processes starve. The values of relevant variables are shown in columns. The array *FBI* has been Curried into $(E \cup T') \to X^D$.

There are at least three ways to ensure fairness:

- Firstly, the verifier executables generated by SPIN have an option for enabling weak fairness. This option may be the one easiest to use.
- Secondly, a *fairness constraint* can be added to the LTL formulae requiring it [24, pp. 49 – 51]. Fairness constraints have the advantage of not changing the model or the arguments passed to the model checker. Properties not requiring fairness, e.g., safety properties, can be left unmodified.
- Thirdly, the model itself can implement a scheduler, for which weak fairness can be proven as an LTL property.

The problem of fairness can also be sidestepped by removing non-progress cycles by adding inter-process coordination (which is in a way comparable to implementing a scheduler). This approach was chosen in the current study. It

has the benefit of having the possibility of reducing the number of reachable states in the model.

Other variants for the PROMELA model were developed in the hope of reducing the number of unfair executions and reachable states in the model by adding extra structure for coordination. These variants explore various ways to improve the communication between Engines and LG, and also between Engines and Terminals. For conciseness, will be explained in terms of changes to the base variant.

4.4 Input Counters Variant

It was mentioned above that the base model suffers from non-progress cycles where an Engine keeps polling the input array. Excessive polling prevents LG and Terminals from carrying out their respective tasks. Adding extra counters shared between LG and Engines is one way to fix this problem.

In the *input counter variant* of the model, LG and Engines share an additional array $Ctr : E \to \{0, 1, \ldots, s\}$, mapping each Engine index $i \in E$ to the number of data points waiting for processing by Engine number i. The array is initialised with $Ctr[i] = 0$ for each $i \in E$. Between lines 5 and 6 in Algorithm 1, the input counter variant LG adds an extra loop where it increments $Ctr[i]$ by one for each $i \in E$ such that $Ctr[i] < s$.

The cyclic mode works similarly as in the base variant. The new loop will be added between lines 6 and 7 in Algorithm 2. The cyclic mode works similarly for variants too, so cyclic mode will not be discussed for the rest of the variants.

The variable *end* is not used in the input counter variant. On line 13 of Algorithm 3, instead of halting if *end* is true, Engine i blocks until $Ctr[i] > 0$. After the call to MULTICAST on line 16 an extra statement $Ctr[i] \leftarrow Ctr[i] - 1$ to decrement the relevant counter is added (staying inside the **if**-block). Otherwise, the input counter variant Engine is the same as the base variant Engine.

4.5 Direct Input Variant

The numerous arrays in the base model led into a large number of reachable states. The state vector size of the base model with two Engines, two Terminals, and three data points per FBI was reported by SPIN to be bytes in size (making the theoretical upper bound for the number of reachable states $2^{8 \cdot 120} = 2^{960} \approx 10^{289}$ or a hundred billionth of a Googol cubed). This led to the creation of a simplified variant of the model which removes the input array.

In the *direct input variant*, LG updates values directly in the FBIs of Engines. More precisely, the array In is dropped and for each $i \in E$ and $k \in D$, $FBI[i, k]$ is initialized to $k + 1$. In line 5 of Algorithm 1, instead of

overwriting $In[k]$ with $In[k] + s + 1$, LG overwrites $FBI[i, k]$ with $FBI[i, k]$ for each $i \in E$. It also marks the data point as dirty by setting its most significant bit (128) to one.

On line 14 of Algorithm 3, instead of checking that $In(k) > FBI[i, k]$, Engine checks checks if $FBI[i, k]$ has been marked dirty, i.e. has its, most significant bit set. On line 15, instead of copying $In[k]$ to $FBI[i, k]$, Engine marks $FBI[i, k]$ clean, i.e. resets its most significant bit to zero.

While the size of the state vector for the direct input variant is 104 bytes, this variant has its downsides. Firstly, 104 bytes is still a huge amount of state information. Secondly, the assumption that LG always updates values in the FBIs of all Engines in synchrony translates into the requirement that DIM must always keep every Engine simultaneously up-to-date. This requirement is not trivial if the network has random delays or disconnects. A small simplification to the model can introduce a great complication to the implementation.

4.6 Direct Input Counters Variant

As the name suggests, the *direct input counters variant* is a combination of the direct input variant and the input counter variant. It merges the changes listed in Chapters 4.4 and 4.5 above. The loops over $i \in E$ are combined into the one loop on lines 5 – 9 in Algorithm 5.

Algorithm 5 Direct input counters variant Load Generator in default mode.

```
 1: process DirectInputLoadGenerator
 2:     local variable  k ← 0
 3:     for z  from  0  to  c − 1 do
 4:         atomic
 5:             for i  from  0  to  e − 1 do
 6:                 FBI[i, k] ← FBI[i, k] + s
 7:                 mark FBI[i, k] as dirty
 8:                 if Ctr[i] < s then
 9:                     Ctr[i] ← Ctr[i] + 1
10:             k ← (k + 1) % s
```

Engine is adjusted accordingly, as shown in Algorithm 6.

4.7 Structured Input Variant

In the base model and the input counter variant, the input array and FBIs use the same indexing. This design has a weakness. When LG writes $v \in X$ to $In[k]$ for some $k \in D$ and Engine i is currently processing index $k + 1$, Engine i has to poll through all indices $k+1, k+2, \ldots s-1$ and then $0, 1, \ldots, k$ before observing $In[k] = v$. As Algorithm 3 shows, Engine checks one data

Algorithm 6 Direct input counters variant Engine i.

```
1: process DirectInputEngine(i)
2:     local variable k ← 0
3:     repeat
4:         atomic
5:             block until Ctr[i] > 0
6:             if FBI[i, k] is dirty then
7:                 mark FBI[i, k] as clean
8:                 Multicast(i, k, FBI[i, k])
9:                 Ctr[i] ← Ctr[i] − 1
10:                k ← k + 1 % s
```

point per execution of its atomic sequence. Other non-blocking processes will be interleaved by SPIN in all possible ways while Engine is polling. This leads into an enormous number of new reachable, but uninteresting, states.

Polling is not the only way to interpret how DIM publish-subscribe connections with client-side callbacks work. In fact, according to Gaspar et al. [25], one of the requirements of DIM is that

> "[a] process should be not [sic] have to poll at regular intervals in order to find out that something changed, it should be informed when it changes".

As only one data point at a time needs to be transmitted, polling can be avoided by sending the index of the data point along with the data point itself.

The input pairs or *structured input variant* of the model shares only one index-value pair between LG and each Engine instance. LG uses an additional global array $\widehat{In} : E \to D \times X$ in variant, which is initialised with pairs $(0, \texttt{NONE})$ in every index $i \in E$. Between lines 5 and 6 in Algorithm 1, the structured input variant LG runs an additional loop over $i \in E$ where it overwrites $\widehat{In}[i]$ with $(k, In[k] + s + 1)$.

Instead of reading from In, Engine reads from $\widehat{In}$. It gets both the index and the value of the data point from this new array. Algorithm 7 shows how Engine works in this variant. As a notational convenience, line 4 uses functional style pattern matching defining new local variables k and v that refer to the first and the second component of $\widehat{In}[i]$ respectively. Because LG updates $\widehat{In}$ atomically (and because in PROMELA, changes to shared variables are immediately visible for all processes), after line 5, Engine has got the updated values for both k and v.

The structured input variant differs in its approach from the direct input and input counters variants so much that no attempt to combine structured input with either of the two variants was made. It can be considered as a

Algorithm 7 Structured input variant Engine i.

```
1: process StructuredInputEngine(i)
2:     repeat
3:         atomic
4:             let (k, v) := \widehat{In}[i]
5:             block until v ≠ NONE
6:             FBI[i, k] ← v
7:             \widehat{In}[i] ← (k, DC)
8:             Multicast(i, k, FBI[i, k])
```

more sophisticated version of the direct input variant that simply does not need additional counters.

4.8 Structured Input and Output Variant

All the variants discussed so far have explored different interpretations of the network connections between LG and Engines, which represents the DIM side of ADAPOS. All of the five variants above share exactly the same code for the procedure Multicast in Algorithm 3 and for Terminals as well, which is sketched in Algorithm 4. The *structured input and output variant* is based on the structured input variant, but it changes the way how Engines send data to Terminals.

In this variant, the array Out is replaced with an array $\widehat{Out} : E \times T \to D \times X$. Compared to the procedure Multicast of Algorithm 3, Engine inspects the second component of $\widehat{Out}[i, j]$ instead of $Out[i, j, k]$ on line 5. Instead of writing v or DC to $Out[i, j, k]$, Engine now writes (k, v) or $(0, \texttt{DC})$ to $\widehat{Out}[i, j, k]$ on lines 6 and 8 respectively. The choice of the first component does not matter if the second component is DC, because Terminal will determine whether or not there is a disconnect by inspecting the second component first.

Just as in the structured input variant, Engine i uses the pair $\widehat{In}[i]$ to determine which data point received an update. Likewise, Terminal j connected to Engine i uses the pair $\widehat{Out}[i, j]$ to do the same. Neither Engines nor Terminals need to do any polling in the structured input and output variant.

As mentioned in Chapter 4, the model behaves “approximately” as if the 0MQ HWM parameter was 1. In the first five variants discussed above, every data point at index $k \in D$ has its own quota of at most one message in transit between an Engine and a Terminal. These variants of the Promela model actually behave as if there were s parallel 0MQ connections between Engine and Terminal. This is the approximation that made building the model a little bit easier, but which sacrificed accuracy.

Real ADAPOS Engines and Terminals share only one 0MQ connection at

a time. In this respect, the structured input and output variant of the model is more realistic, because there can be at most one pair $(k, v) \in D \times X$ in transit at a time. The same argument applies to the last two variants.

4.9 Input Channels Variant

All of the six variants discussed so far use shared memory for communicating. The following slogan from Effective Go[30] has been credited to Rob Pike:

> "Do not communicate by sharing memory; instead, share memory by communicating."

Channels were introduced by Tony Hoare in his seminal paper bearing the name of the process algebra Communicating Sequential Processes (CSP) [32]. Channels are an alternative mechanism for communicating between processes. The slogan above can be interpreted to encourage the use of channels instead of shared memory. There are similarities between PROMELA and CSP, including not only channels but also guarded commands that were introduced by Dijkstra [18]. The Chapter 7 in Ben-Ari's book [6] explains how channels can be used in PROMELA.

Channels pass messages between processes in a *first in, first out* fashion; messages are read from a channel in the same order they were written into it. Many channels buffer messages. In this sense, ADAPOS as a whole is one large distributed channel. It makes sense to attempt modelling ADAPOS by using channels as a communication primitive.

A statement $c\,!\,m$ writes the message m to channel c while $c\,?\,m$ reads a message from channel c into m. A channel has a finite capacity. If c is at maximum capacity, then $c\,!\,m$ blocks until another process reads from c, freeing capacity for m. If c is empty, then $c\,?\,m$ blocks until some process sends a message to c. In both cases, when blocking ends, both processes move together, performing a handshake action, as explained in Chapter 3.1.

Channels with capacity zero, or *rendezvous channels* [6, Ch. 7.2] are a special case. With rendezvous channels, every read or write blocks until another process performs the corresponding write or read. When the read and write operations happen, they occur simultaneously as a handshake action, resulting in a value being copied from one process to the other. This is the semantics of Hoare's original channel concept [32].

In PROMELA unlike in CSP [32], channels can be referred to using variables, like c above. This makes it possible to have multiple processes reading from and writing to the same channel. Even the same process can write to and read from the same channel. The channel variants of the PROMELA model of ADAPOS have only one process writing to and one process reading from each channel.

[30] https://golang.org/doc/effective_go#sharing

The *input channels* variant of the Promela model of ADAPOS introduces an array $\widetilde{In}$ of channels indexed over E. Each channel has capacity for one message $(k, v) \in D \times X$. LG has an extra variable $w \in D \times X$. Algorithm 8 shows the input channels variant of LG. Unlike with arrays, subsequent writes to a channel do not overwrite its previous contents. The **if**-statement on line 7 takes care of removing the previously sent message from the channel $\widetilde{In}[i]$ if it's still there. In Promela, LG has to perform two checks, one to see if the channel is full and another to see if it is empty, instead of just one check and an **else** branch.

Algorithm 8 Input channels variant Load Generator in default mode.

1: **process** ChanneledLoadGenerator
2: **local variable** $k \leftarrow 0$
3: **for** z **from** 0 **to** $c - 1$ **do**
4: **atomic**
5: $In[k] \leftarrow In[k] + s + 1$
6: **for** i **from** 0 **to** $e - 1$ **do**
7: **if** full $\widetilde{In}[i]$ **then** $\widetilde{In}[i] \mathbin{?} w$
8: $w \leftarrow (k, In[k])$
9: $\widetilde{In}[i] \mathbin{!} w$
10: $k \leftarrow (k + 1) \,\%\, s$

The input channels variant of Engine does not need extra variables such as *end* to decide when it may proceed. Instead, Engine i relies on its input channel $\widetilde{In}[i]$ to do the blocking when there are no incoming data points. Engine has its own local copy of the variable w. Instead of the **let** expression in line 4 in Algorithm 7, Engine first performs the read operation $\widetilde{In}[i] \mathbin{?} w$. Line 5 is replaced with **let** $(k, v) := w$.

4.10 Input and Output Channels Variant

The last variant is the *input and output channels variant.* Based on the input channels variant, it adds another channel $\widetilde{Out}$ for each pair $(i, j) \in E \times T$. Algorithm 9 shows the version of the multicast procedure for this variant. When Engine i disconnects from Terminal j, it removes the message that was currently being transmitted in $\widetilde{Out}[i, j]$ and sends the byte `DC`.

Algorithm 10 shows the logic of Terminal j in this variant of the model.

4.11 Spin to LaTeX Converter

Besides the models, a model checking project may need auxiliary scripts and tools for generating properties, running the model checker and analysing the results. A tool named *Spin2Latex* was created for scraping important numbers from the output of the Spin verification runs. It reads text files, and

Algorithm 9 The multicast procedure for Engine i that uses channels.

```
1: procedure ChanneledMulticast(i, k, v)
2:     for j in 0 to t − 1 do
3:         flip coin
4:         if heads then
5:             if ¬(full Õut[i, j]) then
6:                 Õut[i, j] ! (k, v)
7:         else
8:             if full Õut[i, j] then
9:                 Õut[i, j] ? (k, v)
10:            Õut[i, j] ! (0, DC)
```

Algorithm 10 Simplified overview of input and output channels variant Terminal j.

```
1: process Terminal(j)
2:     local variable i ← 0
3:     start: connect(i)
4:     repeat
5:         atomic
6:             block until Out[i, j, k] ≠ NONE
7:             if Out[i, j, k] ≠ DC then
8:                 FBI[j + e, k] ← Out[i, j, k]
9:                 Out[i, j, k] ← NONE
10:            else
11:                i ← (i + 1) % e
12:                goto start
13:            k ← (k + 1) % s
```

produces LaTeX tables, like the ones shown in the Chapter 6. This tool was written in Haskell and it was extended to other output formats besides LaTeX. This tool is publicly available at https://gitlab.com/jllang/spin2latex.

5 The Specification of ADAPOS

The main properties of the model will be discussed next. The purpose of these properties is to formalise the requirements discussed in Chapter 2.3. The properties are what the verifier executable of SPIN verifies.

As a notational shorthand, denote

$$\bigwedge_{i=0}^{n} \varphi(i) := \varphi(0) \land \varphi(1) \land \ldots \land \varphi(n).$$

SPIN does not support indexed conjunctions. Each formula featuring indexed conjunctions has to be hard-coded separately for any particular value n. Save for syntactic differences, the LTL formulae presented in this chapter faithfully reflect their counterparts studied with the SPIN model checker on the PROMELA model of ADAPOS. The LTL properties checked with SPIN in this study were instantiated with, $s = 3$, $e = 2$, and $t = 2$. Recall that s, e, and t stand for service count, event engine count, and terminal count respectively.

5.1 Sanity

In every variant of the PROMELA model of ADAPOS, data points are represented as bytes. Access to scalar values (i.e. Booleans, bytes, and integers) in atomic in PROMELA. Thus, requirement 1. in Chapter 2.3 is trivial for the PROMELA model. The model focuses on requirements 2. – 4. The model compensates the omission of requirement 1. with three other safety properties which will be discussed in this Chapter as well.

In the PROMELA code, the values in the sets X and M are both represented as bytes. Specifically, `DC` $= 126$ and `NONE` $= 127$. A priori, nothing prevents a byte variable representing a value $v \in X$ overflowing and thus unintentionally becoming the reserved constant 126 or 127. An overflow could also result in the bit 128 (see Chapter 4.5) of the variable becoming one when it should remain zero. The first property of the model, called *sanity*, focuses on detecting overflows indirectly.

As mentioned in Chapter 4, all values in the model are bounded above by four compile-time constants. Recall that these constants are service count s, Engine count e, Terminal count t, and event count c. In cyclic mode, LG keeps cycling over three values per data point. On line 6 of Algorithm 2, LG sets the value of data point k to $zs + k + 1$. Since k cycles between 0 and $s - 1$, and z cycles between 0 and 2, the maximum value produced by LG is

$$zs + k + 1 = 2s + (s - 1) + 1 = 3s.$$

Hence, for the cyclic mode, sanity can be expressed as a simple static property of a compile-time constant,

$$3s < 126. \tag{5}$$

For the default mode, deriving the upper bound requires more work. The derivation is discussed in Appendix A. Sanity for the default mode can be stated as the inequality

$$c + s < 126. \tag{6}$$

There are quite many choices for c and s that prevent their sum from overflowing. The correctness of the model does not hinge on sanity, though, because other safety properties can also detect the consequences of overflows. More importantly, in the experiments performed on the model, c and s were chosen to be single-digit numbers.

The proofs for the upper bound for the values produced by LG in Appendix A demonstrate the use of deductive reasoning on pseudocode level. For the actual Promela code, a formal proof would require defining formal semantics [62] for Promela first. Even for small programs, correctness proofs can be quite tedious to produce. There might be hundreds of proofs like the ones presented in Appendix A to be performed for real code. This provides motivation for automated verification.

The Büchi automaton $B_{\neg sanity}$ that accepts the negation of sanity (or insanity) has only two states, the initial non-accepting state s_1 and an accepting state s_2 into which the automaton moves if insanity holds. The state s_2 has a self-loop on every action. If the model is given sane parameters, then $B_{\neg sanity}$ fails to take the first step, rejecting insanity. Otherwise, $B_{\neg sanity}$ moves to the accepting state and loops there forever, accepting insanity.

Model checking is an overkill for such a simple static property. Still, as a property, sanity is a great tool for making sure that the parameters c and s are within the safe range. Unlike sanity, other properties that were studied with the model, have temporal operators and refer to non-constant values. They cannot be verified by checking the initial state alone.

5.2 Safety

Another property which more directly enforces the requirement that values in FBIs must not overflow is the property named vaguely as *safety*. Safety asserts that the values in all indices in the FBIs of Terminals must remain below 126 at all times. Formally,

$$\Box\left(\bigwedge_{j=0}^{t-1}\bigwedge_{k=0}^{s-1} FBI[j,k] < 126\right). \tag{7}$$

For example, when $t = 2$ and $s = 3$, the formula (7) expands to

$$\Box((FBI[0,0] < 126) \wedge (FBI[0,1] < 126) \wedge (FBI[0,2] < 126) \wedge \\ (FBI[1,0] < 126) \wedge (FBI[1,1] < 126) \wedge (FBI[1,2] < 126)).$$

5.3 Disjointness

Consider requirement 2. disjointness, in Chapter 2.3. One one hand, all values of data point k generated by LG are congruent, modulo s. On the other hand, the values generated to data points $k, k' \in D$ are always distinct when $k \neq k'$. ADAPOS processes are not allowed to modify the data point values they store and transfer. Values that were distinct when they left LG must remain distinct when they are stored in the FBIs of Terminals.

This requirement has one exception: Initially, all FBIs are initialised with zeros. Terminals are permitted to have zeros at multiple indices while waiting for updates from Engines. Hence, values in the FBIs of Terminals must stay zero weakly until they become disjoint.

Disjointness can be formalised as

$$\Box \left(\bigwedge_{j=0}^{t-1} \bigwedge_{k=0}^{s-2} \bigwedge_{k'=k+1}^{s-1} \mathit{Null}(j,k,k') \vee \mathit{Disjoint}(j,k,k') \right), \tag{8}$$

where

$$\begin{aligned} \mathit{Null}(j,k,k') &:= \mathit{FBI}[e+j,k] = 0 \wedge \mathit{FBI}[e+j,k'] = 0 \text{ and} \\ \mathit{Disjoint}(j,k,k') &:= \mathit{FBI}[e+j,k] \neq \mathit{FBI}[e+j,k']. \end{aligned}$$

5.4 Checksums

The property named *checksums* refines safety and disjointness. It asserts that for those indices of Terminal FBIs that receive updates, the values must have the correct remainder when divided by s. Recall that each data point $In(k)$ is initialised as $k+1$ and only modified by incrementing it by s. Hence, for FBI index k, the correct remainder is $(k+1) \,\%\, s$. Formally,

$$\Box \left(\bigwedge_{j=0}^{t-1} \bigwedge_{k=0}^{s-1} \mathit{FBI}[e+j,k] = 0 \vee \mathit{FBI}[e+j,k] \,\%\, s = (k+1) \,\%\, s \right), \tag{9}$$

5.5 Causality

The property 3. causality, is interpreted as a requirement that the values stored in the FBIs of Terminals must be less than or equal to the current input values provided to Engines. Formally,

$$\Box \left(\bigwedge_{j=0}^{t-1} \bigwedge_{k=0}^{s-1} \mathit{FBI}[e+j,k] \leq In[k] \right). \tag{10}$$

The direct input and direct input counters variants do not have the array In. In these variants, the property is instead expressed in terms of FBIs

of Engines. Since LG always keeps the FBIs of all Engines synchronised, it does not matter which Engine's FBI is referred to in LTL properties. The only complication is that the most significant bit must be removed by using a bitwise AND mask.

5.6 Monotonicity

The requirement 4. monotonicity, i.e. that Terminals may not change the order of updates, can be expressed in terms of consecutive updates: If value b arrives after value a to a Terminal, then the Terminal must store b after a in its FBI. Both the previous value a and the current value b of a data point needs to be stored in order to express monotonicity.

It was discovered that monotonicity may be temporarily violated when a Terminal connects to a different Engine which has older data points in its FBI. An additional boolean value was added for tracking the freshness of data points. A data point is fresh if and only if it has been received as a part of the FBI sent by an Engine during the connection procedure.

Therefore, FBI entries for Terminals were changed into triples $X \times X \times \{0,1\}$, where the first component stands for the current value, the second component stands for the previous value, and the third component is a Boolean expressing freshness. Initially, $\mathit{FBI}[e+j,k] = (0,0,0)$ for all $j \in T$, $k \in D$. Whenever Terminal j updates the entry (v,u,b) with a new value v' with freshness b', it writes (v',v,b') into $\mathit{FBI}[e+j,k]$. Thus, monotonicity can be formalised as

$$\Box \left(\bigwedge_{j=0}^{t-1} \bigwedge_{k=0}^{s-1} \mathit{Fresh}(j,k) \vee \mathit{Previous}(j,k) \leq \mathit{Current}(j,k) \right), \tag{11}$$

where

$$\begin{aligned} (v,u,b) &:= \mathit{FBI}[e+j,k], \\ \mathit{Fresh}(j,k) &:= b, \\ \mathit{Previous}(j,k) &:= u, \\ \mathit{Current}(j,k) &:= v. \end{aligned}$$

There are three things to note about this formalisation. Firstly, the notion of monotonicity used in this thesis follows a common convention in computer science. By this convention, a sequence $x_1, x_2, x_3, \ldots$ of elements of a preorder $(A, \preceq)$ is called *monotone*, or order-preserving, if $i \leq j$ implies $x_i \preceq x_j$ for all indices i and j. In more precise language, this definition actually defines *isotone* or non-decreasing sequences, whereas monotonicity in mathematics entails *antitone* or order-reversing sequences as well.

Secondly, it is sufficient to only consider pairs of consecutive updates. For example, consider the following two sequences of natural numbers:

$$\sigma_1 = 0, 1, 2, 3, 4, 5, 6, 7, \ldots$$
$$\sigma_2 = 0, 1, 2, 4, 3, 5, 6, 7, \ldots$$

σ_1 is monotone because $n \leq k$ implies $\sigma_1(n) \leq \sigma_1(k)$. On the other hand, $3 \leq 4$, but $\sigma_2(3) = 4 > 3 = \sigma_2(4)$, so σ_2 is not monotone (it's neither order-preserving nor order-reversing). In general, for any sequence σ, it suffices to find an index n such that $\sigma(n) > \sigma(n+1)$ to show that σ is not monotone (in the computer science sense).

Thirdly, the formalisation of monotonicity in Formula (11) only mentions Terminals. This is not an issue. If an Engine violates monotonicity, i.e. changes the ordering of updates and a Terminal honours monotonicity, then the Terminal simply repeats the mistake made by Engine, so the violation will be caught by the model checker. If Engines and Terminals both violate monotonicity in such way that the violations cancel out, then from the perspective of the outside world, nothing bad happened. The purpose of the PROMELA model of ADAPOS is to explore the behaviours visible to the outside world.

5.7 Combined Effect of the Safety Properties

Consider the case $s = 3, c = 3$ in the default mode. The Load Generator will start with values $(1, 2, 3)$, while the ADAPOS Terminals will initially have values $(0, 0, 0)$ in their FBIs. LG will update each data point exactly once. The input that LG produces to Engines will be

$$(1, 2, 3), (4, 2, 3), (4, 5, 3), (4, 5, 6).$$

Table 3 shows the contents of the FBI of a Terminal in hypothetical counterexamples. The last row contains values violating the safety properties for each column while the first four lines contain legal FBIs.

Safety	Disjointness	Checksums	Causality	Monotonicity
(0,0,0)	(0,0,0)	(0,0,0)	(0,0,0)	(0,0,0)
(1,0,0)	(1,0,0)	(0,0,3)	(0,2,0)	(1,2,3)
(1,2,0)	(1,0,3)	(0,2,3)	(1,2,3)	(1,2,3)
(1,2,3)	(1,2,3)	(1,2,3)	(4,2,3)	(4,5,6)
(1,2,126)	(3,2,3)	(1,2,4)	(7,2,3)	(4,5,3)

Table 3: Snapshots of a Terminal's FBI during five hypothetical executions.

All of the five initial fragments in the table would be accepted by all of the five safety properties. Each property focuses on a particular aspect of

safety. Together these properties define which sequences of Terminal FBI values are to be considered valid.

There are two details to consider. Firstly, none of the properties dictate in which order Terminals must receive updates, as long as the values themselves are correct. Secondly, as the monotonicity column hints, the five properties also allow so-called stuttering steps, where the FBI contents are duplicated, as well as the omission of intermediary steps.

6 Results

After numerous iterations of running SPIN and fixing errors in the model and its specification, the final results were obtained with version 6.6 of the model. The properties disjointness and checksums had to be weakened due to a counter-example found by SPIN. The two properties were updated in versions 6.8 and 6.10, with no other changes, and checked again with SPIN.

For eight variants, two modes for LG, ten properties, and four event count values, the total number of runs of the SPIN model checker was 640. The output of the model checker created a dataset of about 30 megabytes of text. This dataset was processed with Spin2Latex version 0.1.4.1. No counter-examples or other errors with default mode LG were encountered for any of the (updated) safety properties that were checked. Due to the large amount of data produced, only the results with default mode LG without LTL properties will be presented in this Chapter.[31]

Tables 4, 5, 6, and 7 show the results for event counts 0, 1, 2, and 3 respectively. States and transitions are in millions, time is in seconds, and memory is in megabytes. For saving space, the words "input" and "output" have been abbreviated as I and O in two rows. One hour time limit was used for all model checking runs, so times exceeding one hour imply timeout. Timeout implies incomplete verification. For event count 0, only two variants timed out while for event count 3, all variants timed out.

Variant	States	Transitions	Time	Memory
Direct Input	0.909	2.964	1.32	772
Direct Input Counters	0.152	0.719	0.47	695
Base*	1,929.597	6,090.928	3609	207,479
Input Counters*	1,732.826	6,377.193	3608	187,475
Structured I & O	0.090	0.270	0.28	689
Input Channels	0.109	0.327	0.32	694
I & O Channels	0.109	0.327	0.32	694
Structured Input	0.090	0.270	0.28	689

Table 4: Results for event count 0. Verification timed out for variants marked with '*'.

The results shown in Tables 4 – 7 seem to suggest that the Structured Input and Output variant has the smallest reachable state space. However, for LTL properties, the Input Channels variant performed better. Figure 17 shows the amount of time and memory consumed for verifying the Input Channels variant without LTL properties using the numbers from Tables 4 – 7. Event count 3 is an exception. As the verification for event count 3 could not be completed in one hour, it had to be rerun without timeout. It

[31] The results are archived at https://doi.org/10.5281/zenodo.4745459

Variant	States	Transitions	Time	Memory
Direct Input	36.238	118.184	63.47	4,870
Direct Input Counters	11.871	60.362	26.78	1,887
Base*	1,933.020	6,101.580	3609	207,827
Input Counters*	1,745.192	6,421.090	3608	188,732
Structured I & O	2.006	7.401	3.37	925
Input Channels	3.013	10.471	4.95	1,095
I & O Channels	1,192.309	4,025.300	2906	194,654
Structured Input	2.746	9.854	4.28	996

Table 5: Results for event count 1. Verification timed out for variants marked with '*'.

Variant	States	Transitions	Time	Memory
Direct Input*	1,922.047	6,056.387	3609	206,706
Direct Input Counters*	1,440.935	6,752.014	3607	157,807
Base*	1,937.755	6,116.348	3609	208,308
Input Counters*	1,693.919	6,235.008	3608	183,521
Structured I & O	110.690	386.985	218	14,745
Input Channels	147.144	495.998	326	23,549
I & O Channels*	1,418.646	4,959.715	3610	227,887
Input Pairs	289.464	914.706	517	36,414

Table 6: Results for event count 2. Verification timed out for variants marked with '*'.

Variant	States	Transitions	Time	Memory
Direct Input*	1,966.080	6,070.566	3609	211,181
Direct Input Counters*	1,634.125	6,897.237	3608	177,443
Base*	1,942.412	6,131.067	3609	208,782
Input Counters*	1,740.866	6,406.162	3608	188,292
Structured I & O*	1,590.470	5,883.783	3609	206,293
Input Channels*	1,552.148	5,518.906	3610	225,897
I & O Channels*	1,413.876	4,940.419	3610	227,178
Input Pairs*	1,866.103	6,052.492	3610	225,379

Table 7: Results for event count 3. Verification timed out for variants marked with '*'.

took 2 hours and 9 seconds and 466,903 megabytes of memory to enumerate the whole reachable state space.

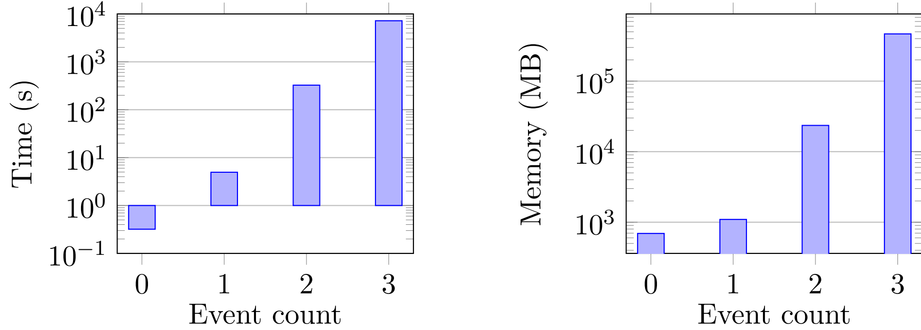

Figure 17: Time and memory consumption for the Input Channels variant.

The memory consumption data behind Figure 17 hints that increasing the event count by one might grow amount the memory needed for verification roughly twenty-fold. The difference in memory consumption between event counts 0 and 1 is an exception, though. The small apparent difference is probably caused by overhead dominating the memory consumption with smaller models.

In addition to having the least state space when combined with LTL properties, Input Channels variant arguably captures the intuitive idea on how DIM and 0MQ behaves the best. For these two reasons, Input Channels variant was chosen for further state space exploration using the bitstate hashing [33] feature of Spin. Bitstate hashing is a probabilistic state space compression technique similar to Bloom filters [19]. It is a good method for quickly covering a large portion of a reachable state space of unknown size [19, p. 368].

Ten additional runs were executed, one without any LTL properties and one for each of the nine LTL properties defined for the model. The verifier executable (Pan) generated by Spin was executed using bitstate hashing with three hash functions, 512 gigabytes of memory reserved for the hash table, one billion entries reserved for the DFS stack and a three-hour timeout. Pan used in total about 580 gigabytes of RAM during these runs. The event count parameter of the model was set to 10 in all runs.

Later, a run without LTL properties but with otherwise same parameters was executed with one week timeout. Pan ran out of time without finding any errors. It managed to cover $1.94 \cdot 10^{11}$ states and $8.11 \cdot 10^{11}$ transitions. If there are deadlocks or other such basic errors in the model, they must be well hidden.

Table 8 summarizes the results. All runs except the ones checking sanity and progress timed out after the three hours. Again, the state and transition counts are in millions. As expected, checking sanity took only one state,

which has been rounded down to 0 in the table. SPIN found a counter-example to the property named progress, which is a liveness property. It took only 1.43 seconds to find this counter-example.

Never Claim	States	Transitions
none*	3,260.836	12,957.467
sanity	0.000	0.000
disjointness*	2,937.172	11,664.587
checksums*	2,945.884	11,700.446
safety*	2,958.879	11,754.421
causality*	2,932.432	11,645.993
monotonicity*	2,946.208	11,701.709
progress*	0.459	1.897
synchrony*	2,911.869	11,563.463
convergence*	2,940.560	11,678.290

Table 8: Results for bitstate hashing with the Input Channels variant and event count 10. Verification timed out for properties marked with '*'.

The main result of this study is that *the model of ADAPOS seems to satisfy its safety properties*, for as far in the reachable state space that could be checked within the time and memory budget available. This should be taken with a grain of salt, though. Bitstate hashing is a sound lossy compression method for state space. *Soundness* means that all counter-examples found with bitstate hashing are legitimate (i.e. bitstate hashing produces no false positives). Due to the lossy nature of the state space compression, some legitimate counter-examples might be missed (possibly leading to false negatives). Thus, like testing, bitstate hashing can only show the presence of errors and not their absence.

It cannot be said that ADAPOS has been proven to be safe. Nonetheless, there is substantial amount of evidence supporting the hypothesis of safety. The volume of state space explored in this study narrows the room in which defects may exist in the specification of ADAPOS.

7 Discussion

This study explored the process of formally modelling a real-life software system. The goal was to show that for a system such as ADAPOS, this non-conventional approach would have its merits and bring in added value. The task turned out to be challenging, but it was a learning experience.

7.1 Lessons Learned

The main lesson learned was that in writing a model, every byte matters. Memory has to used sparingly, perhaps similarly to how programming might have been in the early days of computing. Back then, memory was a scarce and precious resource. Today, there is plenty of memory available. The reason why memory is a bottle neck for model checking is that every bit of memory used has the potential of doubling the size of the state space.

Some of the challenges in the modelling process came from interpreting the DIM and 0MQ protocols. Their published documentation and specifications are unfortunately not precise enough for direct mathematical formalisation. The author of this thesis had to develop new formalisations for these two protocols. The focus of the model presented in this thesis is not on the underlying protocols, but rather on how the distributed system of ADAPOS works on top of them. The DIM and 0MQ protocols deserve their own more detailed models.

7.2 Experiences with Tools and Technologies

As mentioned in the beginning of Chapter 2, a complementary model checker, DIVINE was tried but had to be dropped from this study. The version[32] of DIVINE tried first seemed to have some technical issue that caused a segmentation violation when the memory usage reached about 272 GB. The possibility of the segmentation violations being caused by the computing platform was not ruled out, though. Newer versions of DIVINE could not be compiled, installed or run in the computing environments offered by University of Helsinki and CSC. An issue[33] has been reported to the developers of DIVINE.

What comes to the three main advantages of model checking [17, p. 4], the promise of fully automating model checking while doing so without special expertise requirements still remains to be delivered. Deploying DIVINE was not a trivial task in the previous study [49, Ch. 7.2]. It took many weeks of work. In the current study, it took months of work to make the PROMELA models of the current work small enough to be verified with SPIN.

[32] 4.1.20+2018.12.17

[33] https://divine.fi.muni.cz/trac/ticket/113

An extra level of model engineering challenge was caused by the design decision of storing the different variants of the model in different branches of the Git version control system that was used in this project. Merging different branches for transferring changes was found to be problematic. If branch A contains changes to lines which have not been changed in branch B, then apparently merging A to B silently overwrites the lines in B that were changed in A. In many occasions this led into unwanted changes being introduced in addition to those changes that were actually meant to be merged.

Cherry-picking commits instead of merging branches seemed to circumvent this problem, but resulted in many duplicated commits on separate branches. As each commit would have to be cherry-picked individually in each relevant branch, this increased the total workload. In retrospect, it might have been a better idea to represent different variants using just a single Git branch with CPP macro guards isolating the variants from each other. On the other hand, CPP is known to create its own set of problems.

The syntax of Promela is quite primitive. It lacks modern comforts such as subroutines/functions, classes/typeclasses, modules, and so on. The type system of Promela is likewise very weak and it seems that many problems that could be caught in compile time are delayed into runtime. This translates into the need for the users to build up their own toolchains and scripts to support larger and more systematic modelling efforts. Also, the output from the verifier executable (Pan) generated by Spin is not in a standardised machine-readable output which again necessitates writing custom tools and scripts to scrape the relevant information from the output.

7.3 Related Work

The earlier article [48] presented the high level design and requirements of ADAPOS. It also contains simulation results focusing on measuring the performance. The conclusion [48, p. 485] of was that

> "ADAPOS turns out to meet its specifications."

The work presented in this thesis supports this claim for safety properties by providing considerable amount of additional evidence obtained with model checking.

As pointed out in Chapter 1, the part of the ADAPRO framework running persistently has been investigated in another study [49]. The topic and the methods of that study and the current work are closely related. However, applying model checking on specification level to ADAPOS turned out to be more challenging than it was with ADAPRO. A factor contributing to the greater difficulty of model checking the specification of ADAPOS is that ADAPOS deals with data whereas ADAPRO focuses merely on control. ADAPOS turned out to be more challenging both in terms of the effort

spent in writing the model and the computational cost of verifying properties of the model. As Baier and Katoen [2, p. 15] point out, model checking generally is better suited for verifying control-intensive rather than data-intensive systems. The empirical experiences of the author of this thesis are in agreement with that observation.

There exists prior work on formally verification of systems at CERN, such as the CMS control system case study [37]. The system under verification in that study is a relative of ALICE DCS. The process algebra-based approach with bounded model checking demonstrates a flavour of model checking different than enumerative model checking that was discussed in this thesis.

As mentioned in Chapter 1, ADAPOS is the successor of the ALICE offline Shuttle [68] system. The article also mentions conditions data, which is the kind of data both of these systems deal with. The Shuttle system helped to identify the problem that ADAPOS tries to solve.

Another piece of technology connected to CERN is the SMI++ framework [23]. It has been used for building control systems, including ALICE DCS [14, 15]. Like ADAPRO, SMI++ uses automata. In fact, ADAPRO has taken a little bit of influence from it. Even though not discussed in this thesis, the FSM mechanism is an important internal technology of ADAPOS.

Cloud technologies such as Apache Kafka, Apache Zookeeper and etcd can be used for similar data pipelining purposes as ADAPOS. Compared to these technologies, ADAPOS is much simpler, with less than 3000 lines of C++14 source code according to David A. Wheeler's tool sloccount[34]. ADAPOS has been confirmed empirically [48] to offer a satisfactory level of real-time performance in the use case it serves.

7.4 Conclusion

In software engineering, the distinctions between specification, design, and implementation of software systems tend to be fuzzy. A few lines of text written in natural language or a handful of diagrams might often be all the specification available for a program. The typical concern of a software development organisation is the time to market. Specification and verification are often seen more as hindrances than essential parts creating value to the project.

The methods in this thesis diverge from traditional software engineering conventions. The original plan was to verify both the specification and the implementation of ADAPOS with respect to both safety and liveness properties with event count 10 or more. These goals turned out to be quite ambitious. The amount of work needed for meeting them was found to be more than what fits in a master's thesis.

Fortunately, Partial success was achieved and deeper understanding on

[34] https://dwheeler.com/sloccount/

the specification was obtained. The author learned a great deal more on modelling software systems. Therefore, the author considers this study to be successful. It seems that employing formal methods is feasible and beneficial indeed.

Nevertheless, it seems that the user experience of model checkers such as Spin and DIVINE still have a long way to go before reaching the level of usability that professional software engineers are rightfully expecting. Usability and quality of life improvements are not mere exercise handouts, but crucial steps toward popularisation of formal methods. It is understandable that researchers cannot afford putting too much effort in this frontier. Their job is not to make polished products ready for commercialisation, but rather to explore the science underlying such tools and tools as well as concepts that do not exist yet. Still, more collaboration between academia and industry in this matter would surely benefit both parties.

7.5 Future Prospects

The possibilities for future work are numerous. The correctness of individual ADAPOS processes on implementation level remains an open research question. Completing the model checking of ADAPOS for liveness properties by finding the suitable fairness assumptions and possible improvements to the model is another important research question left open by the current study.

The eight particular combinations of different input and output mechanisms investigated in this thesis are not the only possible interpretations how the network layer beneath ADAPOS works. The input and output side of ADAPOS are orthogonal. For instance, a variant with direct input and output channels could have been investigated, but the way how different variants were constructed, was not scalable and sustainable. A better methodology for maintaining multiple alternative variants would be a valuable direction for future research. Formalising and verifying the DIM or 0MQ protocols is a also an open direction.

A modelling language with multiple inheritance (e.g., by object-oriented classes or functional type classes) could be handy for maintaining many alternative variants of a model. Subroutines would be very useful as well. These changes could make model checking much more complicated, similarly to how interprocedural analyses are more complicated than intraprocedural analyses [52, Ch. 2.5]. Model checking algorithms that support complex modelling language may require complex proofs of correctness.

Adding a support for bounded quantifiers or indexed connectives could be a low-hanging fruit. This could be done by simple preprocessing, i.e. by defining $\forall i \in \{n, n+1, \ldots, n+k\}.P(i)$ to be a macro expanding to $P(n) \wedge P(n+1) \wedge \ldots . \wedge P(n+k)$. Similarly, existential quantification could be expressed using many disjunctions.

Debugging models with Spin and its verifier executable can be frustrating

for beginners. There could be a tool for extracting information, such as the values of all variables in each state, from traces. Traces and states could be displayed in adjacent panels similarly to how debugger GUIs work.

7.6 Acknowledgements

The author of this thesis wishes to thank the Finnish Grid and Cloud Infrastructure (FGCI) for supporting this project with computational and data storage resources during the early stages of this project. The author of this thesis is also grateful for CSC – IT Center for Science, Finland, whose Puhti computing cluster provided resources for producing the final results discussed in this thesis.

References

[1] Abelev, Betty *et al.*: *Upgrade of the ALICE Experiment: Letter of Intent*. (CERN-LHCC-2012-012. LHCC-I-022. ALICE-UG-002), August 2012. https://cds.cern.ch/record/1475243.

[2] Baier, Christel and Katoen, Joost Pieter: *Principles of Model Checking*. MIT Press, April 2008, ISBN 978-0-262-02649-9. https://mitpress.mit.edu/books/principles-model-checking.

[3] Ball, Thomas, Cook, Byron, Levin, Vladimir, and Rajamani, Sriram K.: *SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft*. In Boiten, Eerke A., Derrick, John, and Smith, Graeme (editors): *Integrated Formal Methods*, pages 1–20, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg, ISBN 978-3-540-24756-2. https://doi.org/10.1007/978-3-540-24756-2_1.

[4] Baranová, Zuzana, Barnat, Jiří, Kejstová, Katarína, Kučera, Tadeáš, Lauko, Henrich, Mrázek, Jan, Ročkai, Petr, and Štill, Vladimír: *Model Checking of C and C++ with DIVINE 4*. In *Automated Technology for Verification and Analysis*, volume 10482 of *LNCS*, pages 201–207. Springer, 2017. https://doi.org/10.1007/978-3-319-68167-2_14.

[5] Barnat, Jiři, Brim, Luboš, and Ročkai, Petr: *Towards LTL Model Checking of Unmodified Thread-Based C & C++ Programs*. In Goodloe, Alwyn E. and Person, Suzette (editors): *NASA Formal Methods*, pages 252–266, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg, ISBN 978-3-642-28891-3. https://doi.org/10.1007/978-3-642-28891-3_25.

[6] Ben-Ari, Mordechai: *Principles of the Spin Model Checker*. Springer, 2008, ISBN 9781846287701. https://dx.doi.org/10.1007/978-1-84628-770-1.

[7] Bessey, Al, Block, Ken, Chelf, Ben, Chou, Andy, Fulton, Bryan, Hallem, Seth, Henri-Gros, Charles, Kamsky, Asya, McPeak, Scott, and Engler, Dawson: *A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World*. Communications of the ACM, 53(2):66–75, February 2010, ISSN 0001-0782. https://doi.org/10.1145/1646353.1646374.

[8] Bowen, J.P. and Hinchey, M.G.: *Ten commandments of formal methods*. Computer, 28(4):56–63, 1995. https://doi.org/10.1109/2.375178.

[9] Bruschi, Danilo, Di Pasquale, Andrea, Ghilardi, Silvio, Lanzi, Andrea, and Pagani, Elena: *Formal Verification of ARP (Address Resolution Protocol) Through SMT-Based Model Checking - A Case Study*. In Polikarpova, Nadia and Schneider, Steve (editors): *Integrated Formal Methods*, pages 391–406, Cham, 2017. Springer International Publishing, ISBN 978-3-319-66845-1. https://doi.org/10.1007/978-3-319-66845-1_26.

[10] Buxton, J. N. and Randell, B. (editors): *Software Engineering Techniques*. Report on a conference sponsored by NATO Science Commitee, Rome, Italy, April 1970. http://homepages.cs.ncl.ac.uk/brian.randell/NATO/.

[11] Cadar, Cristian, Dunbar, Daniel, and Engler, Dawson: *KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs*. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, USA, 2008. USENIX Association. https://llvm.org/pubs/2008-12-OSDI-KLEE.html.

[12] Chandra, Satish, Godefroid, Patrice, and Palm, Christopher: *Software Model Checking in Practice: An Industrial Case Study*. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 431–441, New York, NY, USA, 2002. Association for Computing Machinery, ISBN 158113472X. https://dx.doi.org/10.1145/581339.581393.

[13] Chen, Zhe, Gu, Yi, Huang, Zhiqiu, Zheng, Jun, Liu, Chang, and Liu, Ziyi: *Model Checking Aircraft Controller Software: A Case study*. Software: Practice and Experience, 45(7):989–1017, 2015. https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2242.

[14] Chochula, P., Augustinus, A., Bond, P. M., Lechman, L. M., Rosinský, P., Kurepin, A. N., and Pinazza, O.: *Operational Experience with the ALICE Detector Control System*. Proc. of International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'13),

San Fransisco, CA, USA, pages 1485–1488, 2013. https://accelconf.web.cern.ch/ICALEPCS2013/papers/frcoaab07.pdf.

[15] Chochula, P., Jirden, L., Augustinus, A., de Cataldo, G., Torcato, C., Rosinsky, P., Wallet, L., Boccioli, M., and Cardoso, L.: *The ALICE Detector Control System.* IEEE Transactions on Nuclear Science, 57(2):472–478, 2010. https://doi.org/10.1109/TNS.2009.2039944.

[16] Clarke, Edmund M., Henzinger, T. A., Veith, Helmut, and Bloem, Roderick P. (editors): *Handbook of Model Checking.* Springer, 2018, ISBN 978-3-319-10574-1. https://doi.org/10.1007/978-3-319-10575-8.

[17] Clarke, Edmund M., Henzinger, Thomas A., Veith, Helmut, and Bloem, Roderick P.: *Introduction to Model Checking.* In Clarke, Edmund M., Henzinger, Thomas A., Veith, Helmut, and Bloem, Roderick P. (editors): *Handbook of Model Checking.*, pages 1–26. Springer International Publishing, Cham, 2018, ISBN 978-3-319-10574-1. https://doi.org/10.1007/978-3-319-10575-8_1.

[18] Dijkstra, Edsger W.: *Guarded Commands, Nondeterminacy and Formal Derivation of Programs.* Communications of the ACM, 18(8):453–457, August 1975, ISSN 0001-0782. https://doi.org/10.1145/360933.360975.

[19] Dillinger, Peter C. and Manolios, Panagiotis: *Bloom Filters in Probabilistic Verification.* In Hu, Alan J. and Martin, Andrew K. (editors): *Formal Methods in Computer-Aided Design*, pages 367–381, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg, ISBN 978-3-540-30494-4. https://doi.org/10.1007/978-3-540-30494-4_26.

[20] Finkel, Alain and Schnoebelen, Philippe: *Well-structured Transition Systems Everywhere!* Theoretical Computer Science, 256(1):63–92, 2001, ISSN 0304-3975. https://www.sciencedirect.com/science/article/pii/S030439750000102X, Special Issue on Infinite-State Systems.

[21] Fischer, Michael J., Lynch, Nancy A., and Paterson, Michael S.: *Impossibility of Distributed Consensus with One Faulty Process.* Journal of the ACM, 32(2):374–382, April 1985, ISSN 0004-5411. https://doi.org/10.1145/3149.214121.

[22] Fitzgerald, John, Bicarregui, Juan, Larsen, Peter Gorm, and Woodcock, Jim: *Industrial Deployment of Formal Methods: Trends and Challenges.* In Romanovsky, Alexander and Thomas, Martyn (editors): *Industrial Deployment of System Engineering Methods*, pages 123–143, Berlin,

Heidelberg, 2013. Springer Berlin Heidelberg, ISBN 978-3-642-33170-1. https://doi.org/10.1007/978-3-642-33170-1_10.

[23] Franek, B. and Gaspar, C.: *SMI++ Object Oriented Framework for Designing and Implementing Distributed Control Systems.* In *IEEE Symposium Conference Record Nuclear Science 2004.*, volume 3, pages 1831–1835, 2004. https://doi.org/10.1109/NSSMIC.2004.1462600.

[24] Gan, Xiang, Dubrovin, Jori, and Heljanko, Keijo: *A Symbolic Model Checking Approach to Verifying Satellite Onboard Software.* Science of Computer Programming, 82:44–55, 2014, ISSN 0167-6423. https://www.sciencedirect.com/science/article/pii/S0167642313000658, Special Issue on Automated Verification of Critical Systems (AVoCS'11).

[25] Gaspar, C., Dönszelmann, M., and Charpentier, Ph.: *DIM, a Portable, Light Weight Package for Information Publishing, Data Transfer and Inter-Process Communication.* Computer Physics Communications, 140(1):102–109, 2001, ISSN 0010-4655. https://www.sciencedirect.com/science/article/pii/S0010465501002600, 11th International Conference on Computing in High-Energy and Nuclear Physics (CHEP 2000).

[26] Gerth, Rob, Peled, Doron, Vardi, Moshe Y., and Wolper, Pierre: *Simple On-the-Fly Automatic Verification of Linear Temporal Logic.* In Dembiński, Piotr and Średniawa, Marek (editors): *Protocol Specification, Testing and Verification XV: Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification, Warsaw, Poland, June 1995*, pages 3–18, Boston, MA, 1996. Springer US, ISBN 978-0-387-34892-6. https://doi.org/10.1007/978-0-387-34892-6_1.

[27] Gilbert, Seth and Lynch, Nancy: *Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services.* SIGACT News, 33(2):51–59, June 2002, ISSN 0163-5700. https://doi.org/10.1145/564585.564601.

[28] Godefroid, Patrice, Klarlund, Nils, and Sen, Koushik: *DART: Directed Automated Random Testing.* SIGPLAN Notices, 40(6):213–223, June 2005, ISSN 0362-1340. https://doi.org/10.1145/1064978.1065036.

[29] Godefroid, Patrice, Levin, Michael Y., and Molnar, David: *Automated Whitebox Fuzz Testing.* In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*, November 2008. https://www.microsoft.com/en-us/research/publication/automated-whitebox-fuzz-testing/.

[30] Goel, Shilpi, Slobodova, Anna, Sumners, Rob, and Swords, Sol: *Verifying X86 Instruction Implementations*. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 47–60, New York, NY, USA, 2020. Association for Computing Machinery, ISBN 9781450370974. https://doi.org/10.1145/3372885.3373811.

[31] Hinchey, M. and Coyle, L.: *Evolving Critical Systems: A Research Agenda for Computer-Based Systems*. In *2010 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems*, pages 430–435, March 2010. https://doi.org/10.1007/978-3-319-10575-8.

[32] Hoare, Charles Anthony Richard: *Communicating Sequential Processes*. Commununications of the ACM, 21(8):666–677, August 1978, ISSN 0001-0782. https://doi.org/10.1145/359576.359585.

[33] Holzmann, Gerard J.: *An Analysis of Bitstate Hashing*. In Dembiński, Piotr and Średniawa, Marek (editors): *Protocol Specification, Testing and Verification XV: Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification, Warsaw, Poland, June 1995*, pages 301–314, Boston, MA, 1996. Springer US, ISBN 978-0-387-34892-6. https://doi.org/10.1007/978-0-387-34892-6_19.

[34] Holzmann, Gerard J.: *The Model Checker SPIN*. IEEE Transactions on Software Engineering, 23:279–295, May 1997. https://doi.org/10.1109/32.588521.

[35] Holzmann, Gerard J.: *Economics of Software Verification*. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, pages 80–89, New York, NY, USA, 2001. ACM, ISBN 1-58113-413-4. https://doi.acm.org/10.1145/379605.379681.

[36] Holzmann, Gerard J.: *Explicit-State Model Checking*. In Clarke, Edmund M., Henzinger, Thomas A., Veith, Helmut, and Bloem, Roderick (editors): *Handbook of Model Checking*, pages 153–171. Springer International Publishing, 2018, ISBN 978-3-319-10575-1. https://doi.org/10.1007/978-3-319-10575-8_5.

[37] Hwong, Yi Ling, Keiren, Jeroen J.A., Kusters, Vincent J.J., Leemans, Sander, and Willemse, Tim A.C.: *Formalising and Analysing the Control Software of the Compact Muon Solenoid Experiment at the Large Hadron Collider*. Science of Computer Programming, 78(12):2435–2452, 2013, ISSN 0167-6423. https://www.sciencedirect.com/science/

article/pii/S0167642312002365, Special Section on International Software Product Line Conference 2010 and Fundamentals of Software Engineering (selected papers of FSEN 2011).

[38] Jhala, Ranjit and Majumdar, Rupak: *Software Model Checking*. ACM Computing Surveys, 41(4), October 2009, ISSN 0360-0300. https://doi.acm.org/10.1145/1592434.1592438.

[39] Kaivola, Roope, Ghughal, Rajnish, Narasimhan, Naren, Telfer, Amber, Whittemore, Jesse, Pandav, Sudhindra, Slobodová, Anna, Taylor, Christopher, Frolov, Vladimir, Reeber, Erik, and Naik, Armaghan: *Replacing Testing with Formal Verification in Intel® Core™ i7 Processor Execution Engine Validation*. In Bouajjani, Ahmed and Maler, Oded (editors): *Computer Aided Verification*, pages 414–429, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg, ISBN 978-3-642-02658-4. https://doi.org/10.1007/978-3-642-02658-4_32.

[40] Kant, Philipp, Hammond, Kevin, Coutts, Duncan, Chapman, James, Clarke, Nicholas, Corduan, Jared, Davies, Neil, Díaz, Javier, Güdemann, Matthias, Jeltsch, Wolfgang, Szamotulski, Marcin, and Vinogradova, Polina: *Flexible Formality Practical Experience with Agile Formal Methods*. In Byrski, Aleksander and Hughes, John (editors): *Trends in Functional Programming*, pages 94–120, Cham, 2020. Springer International Publishing, ISBN 978-3-030-57761-2. https://doi.org/10.1007/978-3-030-57761-2_5.

[41] Klein, Gerwin, Elphinstone, Kevin, Heiser, Gernot, Andronick, June, Cock, David, Derrin, Philip, Elkaduwe, Dhammika, Engelhardt, Kai, Kolanski, Rafal, Norrish, Michael, Sewell, Thomas, Tuch, Harvey, and Winwood, Simon: *SeL4: Formal Verification of an OS Kernel*. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery, ISBN 9781605587523. https://dx.doi.org/10.1145/581339.581393.

[42] Kupferman, Orna: *Automata Theory and Model Checking*. In Clarke, Edmund M., Henzinger, Thomas A., Veith, Helmut, and Bloem, Roderick (editors): *Handbook of Model Checking*, pages 107–151, Cham, 2018. Springer International Publishing, ISBN 978-3-319-10575-8. https://doi.org/10.1007/978-3-319-10575-8_4.

[43] Lahtinen, Jussi, Valkonen, Janne, Björkman, Kim, Frits, Juho, Niemelä, Ilkka, and Heljanko, Keijo: *Model Checking of Safety-critical Software in the Nuclear Engineering Domain*. Reliability Engineering & System Safety, 105:104–113, 2012, ISSN 0951-8320. https://www.

sciencedirect.com/science/article/pii/S0951832012000555, ESREL 2010.

[44] Lamport, Leslie: *Proving the Correctness of Multiprocess Programs*. IEEE Transactions on Software Engineering, SE-3:125–143, 1977. https://doi.org/10.1109/TSE.1977.229904.

[45] Lamport, Leslie: *How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs*. IEEE Transactions on Computers, C-28(9):690–691, September 1979, ISSN 0018-9340. https://doi.org/10.1109/TC.1979.1675439.

[46] Lamport, Leslie: *Paxos Made Simple*. ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001), pages 51–58, December 2001. https://www.microsoft.com/en-us/research/publication/paxos-made-simple/.

[47] Leroy, Xavier: *Formal Verification of a Realistic Compiler*. Communications of the ACM, 52(7):107–115, July 2009, ISSN 0001-0782. https://doi.org/10.1145/1538788.1538814.

[48] Lång, John L., Augustinus, André, Bond, Peter M., Chochula, Peter, Lechman, L. Mateusz, Pinazza, Ombretta, and Kurepin, Alexandr N.: *ADAPOS: An Architecture for Publishing ALICE DCS Conditions Data*. In *Proc. of International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'17), Barcelona, Spain, 8-13 October 2017*, number 16 in *International Conference on Accelerator and Large Experimental Control Systems*, pages 482–485, Geneva, Switzerland, January 2018. JACoW, ISBN 978-3-95450-193-9. https://jacow.org/icalepcs2017/papers/tupha042.pdf.

[49] Lång, John L. and Prasetya, I. S. W. B.: *Model Checking a C++ Software Framework: A Case Study*. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 1026–1036, New York, NY, USA, 2019. Association for Computing Machinery, ISBN 9781450355728. https://doi.org/10.1145/3338906.3340453.

[50] Natarajan, V. and Holzmann, Gerard J.: *Outline for an Operational Semantics of PROMELA*. In *The SPIN Verification System. Proceedings of the Second SPIN Workshop 1996., volume 32 of DIMACS. AMS*. American Mathematical Society, 1997.

[51] Newcombe, Chris, Rath, Tim, Zhang, Fan, Munteanu, Bogdan, Brooker, Marc, and Deardeuff, Michael: *How Amazon Web Services Uses Formal*

Methods. Commununications of the ACM, 58(4):66–73, March 2015, ISSN 0001-0782. https://dx.doi.org/10.1145/2699417.

[52] Nielson, Flemming, Nielson, Hanne Riis, and Hankin, Chris: *Principles of Program Analysis*. Springer-Verlag, Berlin, Heidelberg, corrected 2nd edition, 2005, ISBN 3-540-65410-0. https://doi.org/10.1007/978-3-662-03811-6.

[53] Ongaro, Diego: *Consensus: Bridging Theory and Practice*. PhD thesis, Stanford University, August 2014. https://purl.stanford.edu/qr033xr6097.

[54] Peled, Doron: *Partial-Order Reduction*. In Clarke, Edmund M., Henzinger, Thomas A., Veith, Helmut, and Bloem, Roderick (editors): *Handbook of Model Checking*, pages 173–190. Springer International Publishing, 2018, ISBN 978-3-319-10574-1. https://doi.org/10.1007/978-3-319-10575-8_6.

[55] Pike, Lee, Wegmann, Nis, Niller, Sebastian, and Goodloe, Alwyn: *Experience Report: A Do-It-Yourself High-Assurance Compiler*. SIGPLAN Notices, 47(9):335–340, September 2012, ISSN 0362-1340. https://dx.doi.org/10.1145/2398856.2364553.

[56] Pnueli, Amir: *The Temporal Logic of Programs*. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57, October 1977. https://dx.doi.org/10.1109/SFCS.1977.32.

[57] Ringer, Talia, Palmskog, Karl, Sergey, Ilya, Gligoric, Milos, and Tatlock, Zachary: *QED at Large: A Survey of Engineering of Formally Verified Software*. Foundations and Trends® in Programming Languages, 5(2-3):102–281, 2019, ISSN 2325-1107. https://dx.doi.org/10.1561/2500000045.

[58] Rozier, Kristin Y.: *Linear Temporal Logic Symbolic Model Checking*. Computer Science Review, 5(2):163–203, 2011, ISSN 1574-0137. https://www.sciencedirect.com/science/article/pii/S1574013710000407.

[59] Ročkai, Petr: *Model Checking Software*. Doctoral theses, dissertations, Masaryk University, Faculty of Informatics, Brno, 2015. https://is.muni.cz/th/tpopu/.

[60] Rupley, Jeff, King, John, Quinnell, Eric, Galloway, Frank, Patton, Ken, Seidel, Peter Michael, Dinh, James, Bui, Hai, and Bhowmik, Anasua: *The Floating-Point Unit of the Jaguar x86 Core*. In *2013 IEEE 21st Symposium on Computer Arithmetic*, pages 7–16, 2013. https://doi.org/10.1109/ARITH.2013.24.

[61] Sakarovitch, Jacques: *Elements of Automata Theory.* Cambridge University Press, 2009, ISBN 9780521844253. http://search.ebscohost.com/login.aspx?direct=true&db=e000xww&AN=656957&site=ehost-live&scope=site.

[62] Schmidt, David A.: *Programming Language Semantics.* In *Encyclopedia of Computer Science*, pages 1463–1466, GBR, 2003. John Wiley and Sons Ltd., ISBN 0470864125. https://dl.acm.org/doi/10.5555/1074100.1074733.

[63] Sen, Koushik, Marinov, Darko, and Agha, Gul: *CUTE: A Concolic Unit Testing Engine for C.* SIGSOFT Software Engineering Notes, 30(5):263–272, September 2005, ISSN 0163-5948. https://dx.doi.org/10.1145/1095430.1081750.

[64] Valmari, Antti: *The State Explosion Problem.* In Reisig, Wolfgang and Rozenberg, Grzegorz (editors): *Lectures on Petri Nets I: Basic Models: Advances in Petri Nets*, pages 429–528, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg, ISBN 978-3-540-49442-3. https://doi.org/10.1007/3-540-65306-6_21.

[65] Wadler, Philip: *Propositions as Types.* Communications of the ACM, 58(12):75–84, November 2015, ISSN 0001-0782. https://doi.org/10.1145/2699407.

[66] Wilcox, James R., Woos, Doug, Panchekha, Pavel, Tatlock, Zachary, Wang, Xi, Ernst, Michael D., and Anderson, Thomas: *Verdi: A Framework for Implementing and Formally Verifying Distributed Systems.* In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 357–368, New York, NY, USA, 2015. Association for Computing Machinery, ISBN 9781450334686. https://doi.org/10.1145/2737924.2737958.

[67] Wolper, Pierre: *Expressing Interesting Properties of Programs in Propositional Temporal Logic.* In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '86, pages 184–193, New York, NY, USA, 1986. Association for Computing Machinery, ISBN 9781450373470. https://dx.doi.org/10.1145/512644.512661.

[68] Zampolli, Chiara, Carminati, Federico, and Colla, Alberto: *The SHUTTLE: the ALICE Framework for the extraction of the conditions Data.* In *Proceedings of 13th International Workshop on Advanced Computing and Analysis Techniques in Physics Research — PoS(ACAT2010)*, volume 093, page 066, February 2011. https://doi.org/10.22323/1.093.0066.

A Derivation for Sanity in the Default Mode

In this appendix, it will be shown that in the default mode, no overflows in the array In occur, if $c + s < 126$. In what follows, the byte constant byte s will be converted implicitly into a natural number when needed by applying the inclusion mapping from bytes to natural numbers, i.e. by assuming unrestricted precision. Let

- $A \subseteq \mathbb{N}^s$ be an array of natural numbers indexed from 0 to $s - 1$, initialised as $A[k] = k + 1$ for all $k \in D$;
- $p \in \mathbb{N} \setminus \{as | a \in \mathbb{N}\}$ be a positive natural number coprime to s;
- $q = \lfloor p/s \rfloor$ be the quotient of p/s; and
- $r = p - qs \in D$ be the remainder of p/s.

Consider a generalisation of Algorithm 1 that replaces In with A. This generalised version of LG will be referred to as the *Generalised Load Generator* (GLG).

Lemma A.1. For all $s > 1$ and $p > 0$, after p updates, GLG has updated the index $r - 1$ last. The values in the array are

$$A[k] = \begin{cases} p + s + k - r + 1 & \text{if } k \in \{0, 1, \ldots, r - 2\} \\ p + s & \text{if } k = r - 1 \\ p + k - r + 1 & \text{if } k \in \{r, r + 1, \ldots, s - 1\} \end{cases} . \quad (12)$$

Proof. By induction on p.

In the base case, $p = 1$. Thus,

$$q = \lfloor p/s \rfloor = \lfloor 1/s \rfloor = 0,$$

because $1/s < 1$ for $s > 1$. Furthermore,

$$r = p - qs = 1 - 0s = 1.$$

During the first step of the loop on lines 3 – 6 of the generalised version of Algorithm 1, $A[k]$ was incremented by s. The variable k was 0 at this point, so GLG only updated the first value in A, i.e. the value $A[0]$, or $A[r - 1]$. Hence, after the first update,

$$\begin{aligned} A &= (1 + s, 2, 3, \ldots, s) \\ &= (1 + s, 1 + 1 - 1 + 1, 1 + 2 - 1 + 1, \ldots, 1 + (s - 1) - 1 + 1) \\ &= (p + s, p + 1 - r + 1, p + 2 - r + 1, \ldots, p + (s - 1) - r + 1), \end{aligned}$$

so the thesis holds when $p = 1$.

For the inductive case, assume that the thesis holds for some $p \geq 1$. In other words, by assumption, after p updates, GLG has updated the index $r-1$ last and the Equation (12) holds. GLG will update the data point at index $(r-1)+1 = r$ with the value $A[r]+s$ next, on line 5 of the generalised version of Algorithm 1.

By induction hypothesis, the first $r-1$ values in A are

$$(p+s-r+1, p+s+1-r+1, \dots p+s+(r-2)-r+1).$$

$A[r-1]$ can also be written as

$$\begin{aligned} p+s &= p+s+(r-r)+(1-1) \\ &= p+s+(r-1)-r+1. \end{aligned}$$

Thus, for all $k < r$,

$$A[k] = p+s+k-r+1. \tag{13}$$

By induction hypothesis,

$$A[r] = p+r-r+1.$$

The new value will be

$$\begin{aligned} A[r]+s &= (p+r-r+1)+s \\ &= (p+1)+s. \end{aligned} \tag{14}$$

GLG did not change any of the values $A[k]$ for $r < k < s-1$. Hence, for $k \geq r$, the induction hypothesis yields

$$A[k+1] = p+(k+1)-r+1.$$

This can rewritten as

$$A[k] = p+k-r+1, \tag{15}$$

for $r < k < s$.

By the base case, equations (13), (14), and (15), and induction, Lemma A.1 holds for all $p > 1$ coprime to s. ■

Lemma A.1 breaks down when $p = as$, because then

$$q = \lfloor p/s \rfloor = \lfloor as/s \rfloor = a.$$

This implies that

$$r = p - qs = as - as = 0.$$

Now $r-1$ becomes negative. The cases $p = 0$, $p = as$, and $s = 1$ can be covered with another lemma.

Lemma A.2. After $p = as$ updates for some $a \in \mathbb{N}$,

$$A = (1 + p, 2 + p, \ldots, s + p).$$

Proof. After $p = as$ updates, GLG has updated every value exactly $q = \lfloor p/s \rfloor$ times. Since each update increments the updated value by s, it follows that after p updates,

$$A = (1 + qs, 2 + qs, \ldots, s + qs). \tag{16}$$

Since

$$qs = \lfloor p/s \rfloor s = \lfloor as/s \rfloor s = as = p,$$

the claim follows by substituting qs with p in Equation (16). ∎

The two lemmas above can be used for proving an upper bound for the values generated by LG.

Theorem A.3. For the default mode LG, the largest value ever stored in In will be at most $c' + s'$, with c' and s' being the natural numbers corresponding to the bytes c and s respectively.

Proof. Firstly, note that no other process write into In. Especially the largest value will be written by LG. Again, let c and s be converted into natural numbers whenever needed in the following reasoning. In other words, the primes in c' and s' will be dropped.

If $c = as$ for some $a \in \mathbb{N}$, then the claim follows immediately from Lemma A.2. If $c \neq as$ for any $a \in \mathbb{N}$, then Equation (12) holds by Lemma A.1. Consider the cases:

- For $k \in \{0, 1, \ldots, r - 2\}$, the largest value will be

$$\begin{aligned} A[r-2] &= c + s + (r - 2) - r + 1 \\ &= c + s - 1 \\ &< c + s. \end{aligned}$$

- For $k = r - 1$, the value will be $A[r - 1] = c + s$.
- For $k \in \{r, r + 1, \ldots, s - 1\}$, the largest value will be

$$\begin{aligned} A[s-1] &= c + (s - 1) - r + 1 \\ &= c + s - r \\ &< c + s, \end{aligned}$$

  because $c \neq as$ implies $0 < r < s$.

Therefore, the largest value written into A by GLG after c updates will be exactly $c + s$. It is not possible for LG to produce values greater than GLG does, but due to byte overflows, LG could produce smaller values. ∎