AGH

University of Science and Technology in Kraków

Faculty of Electrical Engineering, Automatics, Computer Science and Biomedical Engineering

PhD Dissertation

Methods of generation of transition systems for Alvis language

Author:

Michał Wypych

Supervisor:

Marcin Szpyrka, Ph.D.

Kraków 2021

Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie

Wydział Elektrotechniki, Automatyki, Informatyki i Inżynierii Biomedycznej

ROZPRAWA DOKTORSKA

METODY GENEROWANIA ETYKIETOWANYCH SYSTEMÓW PRZEJŚĆ DLA JĘZYKA ALVIS

AUTOR:

Michał Wypych

PROMOTOR:

prof. dr hab. Marcin Szpyrka

Kraków 2021

I would like to thank my wife for supporting me and motivating me to complete this dissertation.

Moreover, I would like to thank my supervisor, professor Marcin Szpyrka, for his dedication, patience and friendly supervision of my research work.

Abstract

This doctoral dissertation deals with methods of formal verification and modelling of distributed systems in Alvis modelling language. In the distributed computing world, the standard testing methods fall short and are unable to guarantee the correctness of the system. The concurrency problems are inherently difficult to address and do right. Moreover, concurrency bugs may hide under normal operation of the system and only manifest themselves when the system undergoes some extreme rare scenarios. Formal methods are specifically useful in the verification of systems that are composed of concurrent distributed processing units, as they allow exhaustive exploration of all possible scenarios, even the rarest that the distributed system may undergo.

Alvis language was designed as an attempt to create a formal modelling language that would be easy to learn and use for software engineers. To meet that goal, Alvis allows specifying models behaviour in a high-level programming language. Dependencies between distributed units are modelled and visualised in a graphical environment. Moreover, different scenarios of execution can be modelled with different system layers. One of the core goals behind the Alvis project is to enable formal automated verification of the specified models.

Before the start of the research presented in this dissertation, Alvis was mostly a theoretical project. Mathematical foundations were already proposed and a tool allowing graphical modelling of the system was present. However, there was no tool performing automated verification of the designed model, as there was no tool to convert a high-level model into a representation suitable for model checking and verification. One such representation is a Labelled Transition System (LTS).

The main goal of this dissertation was to propose, design and implement a tool that would be responsible for automatic, fast, and unsupervised generation of the Labelled Transition System from the high-level Alvis model description. This goal was achieved by the creation of the Alvis compiler tool that allows the generation of a Labelled Transition System consisting of 10^8 states on a moderate personal computer in a few hours. The compiler performs the generation of the LTS in a two-stage approach, known from similar tools. The first stage is a creation of an implicit representation of the LTS in a general-purpose language – Haskell, known as Intermediate Haskell Representation (IHR). The second stage is a compilation and running of the IHR program to create LTS. That enabled enormous expressiveness and flexibility of the proposed solution.

The goal of the dissertation was achieved successfully and compared with the state-of-the-art model checking tool – Spin. Alvis compiler achieves similar performance as a well-established tool and allows the creation of LTS for arbitrary Alvis models.

Niniejsza rozprawa doktorska dotyczy metod weryfikacji formalnej i modelowania systemów rozproszonych w języku modelowania formalnego Alvis. w dzisiejszym świecie rozproszonych system komputerowych standardowe metody testowania zawodzą i nie są w stanie zagwarantować jakości i poprawności systemu współbieżnego. Problemy ze współbieżnością są z natury trudne do za modelowania i poprawnego rozwiązania z ominięciem problemów. Co więcej, błędy ze współbieżnym lub rozproszonym przetwarzaniem mogą ukrywać się podczas normalnego działania systemu i ujawniać się tylko wtedy, gdy system doświadcza skrajnie rzadkich scenariuszy. Metody formalne są szczególnie przydatne w weryfikacji systemów, które składają się ze współbieżnych lub rozproszonych jednostek przetwarzania, ponieważ umożliwiają wyczerpującą eksplorację wszystkich możliwych scenariuszy, nawet tych najrzadszych, z jakimi może się spotkać system w trakcie działania.

Język Alvis został zaprojektowany jako próba stworzenia formalnego języka modelowania, który byłby łatwy do nauki i użycia dla inżynierów oprogramowania. Aby osiągnąć ten cel, Alvis umożliwia określenie zachowania modeli w języku programowaniu wysokiego poziomu. Zależności pomiędzy rozproszonymi jednostkami są modelowane i wizualizowane w środowisku graficznym. Co więcej, różne scenariusze wykonania mogą być modelowane za pomocą różnych warstw systemowych. Jednym z głównych celów projektu Alvis jest umożliwienie formalnej i automatycznej weryfikacji dowolnych modeli.

Przed rozpoczęciem badań przedstawionych w niniejszej rozprawie Alvis był głównie projektem teoretycznym. Zaproponowano już podstawy matematyczne i udostępniono narzędzie umożliwiające graficzne modelowanie systemu. Brakowało natomiast narzędzia do automatycznej weryfikacji zaprojektowanego modelu, gdyż nie było narzędzia do konwersji model określonego w języku wysokiego poziomu do reprezentacji przystosowanej do weryfikacji modelowej. Jedna z takich reprezentacji to etykietowany system przejść (LTS).

Głównym celem niniejszej rozprawy było zaproponowanie, zaprojektowanie i wdrożenie narzędzia, które odpowiadałoby za automatyczne, szybkie i nienadzorowane generowanie etykietowanych systemów przejść na podstawie wysokopoziomowego opisu modelu w języku Alvis. Cel ten został osiągnięty dzięki stworzeniu kompilatora języka Alvis, który umożliwia generowanie LTS'ów składający się z 10^8 stanów na przeciętnym współczesnym komputerze stacjonarnym. Cały proces jest realizowalny zaledwie w ciągu kilku godzin. Kompilator generuje LTS w dwuetapowym podejściu, znanym z podobnych narzędzi. Pierwszym etapem jest stworzenie niejawnej reprezentacji LTS w języku ogólnego przeznaczenia – Haskell, znanym jako pośrednia reprezentacja w Haskellu (IHR). Drugi etap to kompilacja i uruchomienie programu IHR w celu utworzenia wynikowego grafu LTS. Proponowane podejście cechuje się ogromną elastycznością i wysoką ekspresywnością.

Cel rozprawy został osiągnięty z powodzeniem i porównany z dobrze znanym narzędziem do weryfikacji modelowej – Spin. Kompilator Alvisa osiąga podobną wydajność jak dobrze znane narzędzia.

Contents

Listings

Chapter 1

Introduction

1.1 Motivation

IT systems are currently in everyday use in every branch of industry. In modern cyber era, all aspects of life depend on some less or more complex computer-based system. Everyday operation of people or even the whole economy depends on the correct operation of IT systems. Modern companies build their competitive advantage of delivering useful applications that automates everyday tasks or make people closer. Due to the recent epidemic situation, everyday work in many companies was shifted into cyberspace in a “work from home” setup. It was possible due to last decade advancements in all aspects related to computer science. Right now, the software is controlling a lot of critical and non-critical aspects of our lives. Due to the above facts, correct operation, high quality and reliability of computer systems are paramount aspects that have to be taken into account.

Another aspect besides quality is a reduction of time of delivery to the market. No company can spend infinite time on assuring the quality of their computing systems. The more time is spent on assuring the quality of the IT solution, the more costly the solution is. On the other hand, reports show [175, 160] that cost of fixing or remediation of the errors is growing fast with the development cycle stages: depending on the stage when the error is finally discovered. The latter in the development cycle an error is found, the more expensive it is to fix.

The type of product delivered may change the cost of fixing the bug significantly. The report created by NASA [160] found that if finding an error in requirements phase costs 1 unit then it increases so significantly that the same defect if undetected until operations phase can cost to fix anywhere from 29 units to more than 1500 units for satellites and aerospace products. The Table 1.1, presents aerospace losses in the 90s due to the software failures aggregated by year for just a few examples of catastrophic failures [175].

There are several infamous system failures with tragic consequences that have been analysed and studied in detail over the years. They have become the symbol of why proper software verification is so important

Table 1.1: Aerospace losses in the 90s due to software failures. Adapted from [175]

year	1993	1996	1997	1998	1999
events	Airbus A320	Ariane 5 Galileo [115]	Lewis	Zenit 2	DS-1 Orion 3
		Flight 965 [37]	Pathfinder	Delta 3	Galileo
			USAF Step	Near	Titan 4B
aggregate cost		$640 million	$116.8 million	$255 million	$1.6 billion
loss of life	2	160			
loss of data	YES	YES	YES	YES	YES

and how tiny errors or misconfigurations can have unexpectedly extensive and tragic cost. They have also been analysed scrutinisingly and many quality and testing standards and operation processes were developed to prevent future failures of similar type. Below the non-exhaustive list of frequently cited accidents is mentioned with a short description of a possible source of failure.

- Radiotherapy system Therac-25 – It was a radiation therapy machine used for treating cancer. It used an updated version of the previous device using a new approach, and it was supposed to be safer. However, between 1985 and 1987, there were six accidental releases of radiation of a maximum dose that resulted in the death of three patients. The reason behind those failures was due to several decisions and mistakes. This improved version of the machine was considered safer as the software safe-guards were added. Therefore, the hardware safe-guards were removed as unnecessary. However, due to a race condition in the code, software safe-guards did not always work. The problem was in a counter within a routine which would often overflow, and if the machine's operator set the value when the overflow happened, the software safe-guard would fail. As a result, the patient would get 100 times greater dosage of radiation then he should.
- Patriot Missile – It is a surface-to-air missile system which was used to intercept Iraqi Scud missiles but failed to do so at 25th February 1991 in Dhahran, Saudi Arabia. As a result, 28 soldiers were killed. The direct reason for the failure was a system error where the system internal clock was off by one-third of a second. The clock drift in turn accumulated over time and affected how the system calculated distance. As a result, the intercepting system lost track of the incoming Scud missile that it initially correctly detected. The reason behind the clock drift was an incorrect conversion between an integer timestamp and a floating-point number used later in the calculation of distance.
- Ariane 5 Galileo mission [115] – The rocket was supposed to deliver four satellite systems to research the Earth's magnetosphere on 4th June 1996. The mission failed to achieve orbit and self-destructed in midair in 39th second after launch. No human was injured, but a significant financial loss was made of more than 370 million dollars. The software defect was a problem in the flight control system reused

from previous Ariane 4 flight control system where the problem was undetected due to the lesser acceleration. The flight control system failed with an exception and was unable to provide correct attitude data for the solid boosters. The failure in the flight computer was due to uncaught software exception resulting from 64-bit floating number to 16-bit signed integer conversion resulting in the failure of the flight computer. Those particular calculations were only required for the previous version of the rocket. The system was duplicated, but the main computer and the backup failed at the same moment as both computers were running the same software.

- Mars Climate Orbiter [106] – It was a robotic space probe owned by NASA that was to study Mars and its climate and atmosphere. However, on 23rd September 1999, it flew too close to mars and disintegrated. The orbiter cost $193 million. The direct cause of why the probe flew too close to Mars was due to two different modules in the software using different units for trajectory calculations. The application called SM_FORCES (small forces) provided data in English units of pound-seconds ($lbf \cdot s$) rather than metric system Newtonseconds ($N \cdot s$). The usage of English units was in contradiction with an MSOP Project Software Interface Specification. Subsequent processing of the data from SM_FORCES by the navigation software algorithm underestimated the effect on the spacecraft trajectory by a factor of 4.45, that was the conversion factor from the force in pounds to Newtons.

1.2 Testing

The most promising and frequently used approach to eliminate the software bugs is a software verification through testing and various quality assurance methods that help to keep the risk of failure low and to raise the trust in the software reliability. However, there is some difficulty in defining and measuring software quality. Common attributes include functionality, reliability, usability, efficiency, maintainability, and portability. But these quality metrics are largely subjective and do not support rigorous quantification that could be used to design testing methods for software developers or support information dissemination to consumers. Information problems are further complicated by the fact that even with substantial testing, software developers do not truly know how their products will perform until they encounter real scenarios. Some existing standards require from the software product to be verified through testing or even application of formal methods [31]:

- IEEE Std 1012-2012 [63] – an IEEE standard for system and software verification and validation.
- ISO 12207 [157] – an international standard for software lifecycle processes.
- European Cooperation for Space Standardization (ECSS), e.g. ECSS-E-ST-10-03C – standards for the system engineering implementation requirements for space systems and space products development.
- DO-178C, DO-278A, and DO-333[155] – a standards for the development of avionics, and air traffic management safety-critical software systems.

One of the possible ways to verify software is to perform software testing. By software testing [139, 15], one can understand the process of applying metrics to determine a software product quality. Software testing is the dynamic execution of software and the comparison of the results of that execution against a set of predetermined criteria. “Execution” is the process of running the software on a computer, with or without any form of instrumentation or test control software being present. “Predetermined criteria” signify that the software’s capabilities are known before its execution. What the software does can then be compared against the anticipated results to judge whether the software behaved correctly.

Automated tests can be run after every change in the software, therefore catching bugs at the earliest stages of the development cycle and minimising the costs of removing the bugs. After the software is put into operational use, a maintenance phase begins where enhancements and repairs are made to the software. During this phase, some or all of the stages of software testing will be repeated together with monitoring of the application performance through collecting the logs and tracking various metrics. This can still spot problems before they escalate into severe catastrophic failures. In the report by NIST [175], some effort was put into the classification of various testing methods. Here, a short classification of various testing groups will be presented.

The first class of testing could be called general testing stages. These stages are basic to software testing and currently are present in all contemporary software development. The following layer are considered general software testing stages:

- unit testing – it is a test of the smallest piece of code (a single class, or a single method). The environment for the tested unit is prepared so that all preconditions are met, and then postconditions are checked for that particular unit of code. At unit testing stage an uncaught exception could be spotted.
- integration testing – an integration test checks if units of code that work correctly independently, work together as a whole piece of software. An integration test could address the problem of disagreement in units used in different modules.
- system testing – a computing system as a whole has to be compliant with requirements. System testing addresses that whole software consisting of all the units satisfies all requirements. When the requirement is changed, it should be reflected in the change of at least one system test. Then if the requirement is no longer satisfied at least one system test should fail.
- regression testing – is a process of re-running all implemented unit, integration and system tests in order to make sure that a change in the software that was introduced does not cause a “regression” – introduces back some known defect.

Apart from general testing stages, specialised software testing stages could be defined. They occur less frequently than general software testing stages and are most common for software with well-specified criteria. The following stages are considered specialised software testing stages:

- stress, capacity, or load testing – it is a specialised system test that is intended to check if code together with hardware can compute, process, and respond for the required amount of load, e.g. a number of requests, or amount of data. An example of the software that allows performing stress testing could be Gatling [49].
- error-handling/survivability testing – a group of tests assuring the quality of application error handling. Those test check if application won't crash when processing erroneous or missing data.
- recovery testing – those are the tests that check how the software reacts on the failures and if it can recover from the failure. Some companies develop specialised software like Chaos Monkey [135] whose purpose is to randomly terminates parts of the running system.
- security testing – those are the tests that target the security gaps in the software.
- viral protection testing stage – anti-virus scanning of the running code can detect viruses.

The last group of tests require the user's involvement, and their information technology consultants are active participants at various stages along the software development process. Users generally participate in the following stages:

- usability testing, AB testing – those tests help to find the elements which are hard to use for the human user or compare two alternative solutions to find out which one is better in terms of interaction with the user.
- field or beta testing – those type of tests require the involvement of the actual customers and are one of the last stages before the application is officially released.
- acceptance testing – the process that is performed by the customer or similar entity that assess if the acquired software satisfies customer's requirement.

1.3 Problems of concurrent computation

Concurrent programs are intrinsically challenging to assure its quality through testing. The possible bugs do not manifest themselves in every execution because they may depend on uncontrollable variables like internal scheduling algorithm. Moreover, scheduling may favour the correct sequence of thread execution interleavings during testing and erroneous scheduling may manifest itself only when the program undergoes heavy load in a production environment. In this and next section, examples and classification of the problems that are intrinsically difficult for testing are described.

Lack of usage of any synchronisation mechanism may lead to a *race condition*. A race condition is a situation where the system's cumulative behaviour is dependent on the sequence of operations of other uncontrollable parts of the system. It can lead to bugs when one of the possible sequences of execution of operations is undesired. It is inherently difficult to target this type of bugs in common testing approaches. As well, encountering those type of errors during debugging is rarely achievable. We can consider a simple `x = x + 1;` instruction in a high level language. When the instruction is compiled, it produces several assembly instructions like load from memory, increment, and store to the memory. Suppose two threads would perform those operations. Some scheduling of the instruction may lead to two threads reading the same value from memory before updating it and eventually storing the same value greater by one. Two threads incremented the value, but the final result is greater only by one. So a race condition leads to the lost update.

In order to synchronise concurrent processes or threads, some carefully prepared mechanism have to be used. The application code can use several synchronisation primitives, here only two most popular are described:

- lock – a lock can be acquired or released. However, contrary to the ordinary boolean variable, those two operations are atomic, and therefore there is no possibility of a race condition situation when two threads will acquire the same lock at the same time. When the lock is acquired by one thread, a second thread is suspended until the lock is released.
- mutex – mutexes and locks allow for the same semantics of acquiring and releasing. The difference is the scope where they can be used. Locks are used within the same process for its thread synchronisation, while mutexes are provided by the operating system and allow synchronisation between different processes.
- semaphore – a semaphore is an extension of the lock. If the lock corresponds to the boolean variable, then semaphore corresponds to the integer variable. A semaphore may allow several – a preconfigured number of threads to access a protected resource before the next thread is blocked. Semaphore can be lowered (decrease the internal counter) or raised (increase the internal counter). If the internal counter drops to zero, then lowering the semaphore will lead to thread blocking.

Since the '60s, there was done an extensive research on concurrent computation problems that stemmed, from e.g. research on databases and data stores. A few problems were formulated in an abstract and distilled fashion, which allow focusing on the problem without going into unnecessary implementation details which are not relevant for solving the original problem. The problems helped to catalogue and classify the type of bugs that may appear in concurrent programs.

In this chapter, two classical problems of concurrent computations are presented shortly, and one the problems will be studied more thoroughly in the next section. One of the most well-known problem in concurrent computation can be *Dining philosophers problem* [54, 79]. The problem is stated as follows: five philosophers are sitting at a round table, sharing only five forks that lay on that table. The philosophers think and eat from the bowl of spaghetti. In order to eat, a philosopher needs to acquire two forks, which he shares with two adjacent philosophers. As the number of the resources (here forks) is limited, philosophers compete between each other for them. Let us assume that every philosopher acts in the same manner: firstly, he tries to lift his left fork; secondly, he lifts his right fork; then he eats; finally he puts down the left fork and lastly, the right fork. Therefore, the philosophers may end up in the situation when no one can take any further action (if every philosopher managed to acquire just his left fork and keeps waiting for its right fork, which is already taken). That situation is called a *deadlock*, and it signifies that a system stuck in the state when no progress is possible. The problem is concrete, but in a general case, the analysis of situations which lead to deadlock was studied and formulated as so-called *deadlock conditions* [45]. There are various solutions to the dining philosophers problem, which may lead to a different type of bugs. One of the possible solutions is to break the symmetry of the algorithm and allow one of the philosophers to take the forks in a different order. This solution is easily implementable in practical applications when shared resources posses some integer identifiers. The philosophers (threads) can acquire forks (shared resources) in a way that always a fork with a smaller identifier is taken before the fork with a greater identifier. However, that solution may lead to another bug, which is called *starvation*. The situation is when due to unfortunate coincidence, at least one of the philosopher is never able to acquire his forks and therefore unable to eat (which in the long run leads to literal starvation).

Another problem is *Readers–Writers problem* that was formulated by Courtois [51]. This problem models different classes of access to the same shared resource. Several writers are having the purpose of writing something into a shared component, and several readers try to read from the same shared component. The apparent solution limiting the access to only one active thread (both writer and reader) is sub-optimal as readers do not necessarily have to compete for the access between each other as reading does not change the state of the shared resource and therefore could be optimised to allow several readers accessing at the same time. In the foundation paper, there were proposed two solutions to that problem that optimise two different aspects. One solution that minimises the delay for readers and another one that prioritises writers access were suggested initially. The first solution uses a single semaphore for writing. If a writer needs to access the resource, it lowers the semaphore; writes and raises the semaphore (rasing a semaphore signals availability of the resource). If a reader needs to access the resource, it atomically increments the read count shared variable and checks if that value is equal to one (both operations has to be performed atomically). If the read count is equal to one, then the write semaphore is lowered. Otherwise, there is already some reader accessing the shared resource. Next, the reader is allowed to read. When finished, the reader count is decremented, and if the value drops to zero, then write semaphore is raised with signalling availability of the resource. The later solution requires to use of two separate semaphores, one for reading and another one for writing [51].

Therefore, the usage of synchronisation primitives can make the code safe from race condition situation, but when performed not carefully may lead to the following problems:

- *Deadlock* – A deadlock state is a situation when several processes are waiting for each other to take action and none of the involved processes can progress. The action could be releasing a lock or a message send (e.g. in the bounded buffer problem a resumption action/message).

- *Livelock* – A live lock state is a situation when several processes are changing their states mutually, but none of the processes can progress further. The term was coined by Ashcroft [9]
- *Starvation* – A situation of starvation is when some process is never scheduled and therefore cannot perform any computation because other processes only gain access to the processing unit.
- *Priority inversion* – A priority inversion state is a situation when two processes with higher and lower priorities are executed, and the lower priority process indirectly preempts higher priority process. This situation may appear when the lower priority process acquired the lock that is required to proceed by a higher priority process.

1.4 Producer-Consumer problem

The Producer-Consumer problem also known under the name of the *Bounded–Buffer* problem is an abstract problem in the domain of concurrent process synchronisation. The problem was described in [53] together with the proposed solution based on semaphores introduced in the same paper. The system is composed of two types of processes. The Producer process is intended to compute values that are eventually supposed to be processed by the Consumer process. Those processes communicate by the shared buffer of finite and fixed size. The Producer process is allowed to put the calculated values into the buffer. The Consumer process is allowed to retrieve the values from the buffer. The number of values in the buffer cannot exceed the fixed size of the buffer. Furthermore, the size of the buffer cannot drop below zero. That conditions imply that the producer cannot put another item into the buffer if the buffer is full. Moreover, the Consumer cannot retrieve an item from the buffer if there is no item. The naïve implementation can lead to severe problems that can happen in the system. Let us first discuss the error-prone implementation in Java. The implementation of the Producer and Consumer are presented in Listings 1.1 and 1.2 respectively.

```java
class Producer implements Runnable {
  Buffer buffer;
  public void run() {
    while (true) {
      var item = produceItem();
      buffer.putItem(item);
  }}
  int produceItem() {/*...*/}
}
```

Listing 1.1: Producer implementation

Both processes implement an infinite loop. Within the loop, the item is calculated and put into the shared buffer or retrieved from the buffer and processed. The calculation and processing of item are not relevant for this problem. Therefore, they were simplified as an empty subprocedure.

```
class Consumer implements Runnable {
  Buffer buffer;
  public void run() {
    while (true) {
      var item = buffer.removeItem();
      consumeItem(item);
  }}
  void consumeItem(int item) {/*...*/}
}
```

Listing 1.2: Consumer implementation

However, the most important is the implementation of the buffer shown in Listing 1.3. The implementation of placing a value into the buffer – `putItem`, checks if the buffer is full and then suspends execution of the current thread (Producer). If the buffer is not full or the thread was resumed, the value is stored in the local buffer (the implementation was omitted), and the internal counter of buffer occupation is incremented. The last step is to resume execution of the suspended Consumer if the buffer was empty, and now it contains a value that can be consumed.

```
class Buffer {
  final int BUFFER_SIZE = 3;
  volatile int itemCount = 0;
  volatile Thread consumer;
  volatile Thread producer;

  boolean isFull() { return itemCount == BUFFER_SIZE;}
  boolean isEmpty() { return itemCount == 0; }

  void putItem(int item) {
    if (isFull()) {
      producer = Thread.currentThread();
      producer.suspend();
    }
    putItemIntoBuffer(item);
    itemCount = itemCount + 1;
    if (itemCount == 1) {
      consumer.resume();
  }}

  int removeItem() {
    if (isEmpty()) {
      consumer = Thread.currentThread();
      consumer.suspend();
    }
    var item = removeItemFromBuffer();
    itemCount = itemCount – 1;
28      if (itemCount == BUFFER_SIZE − 1) {
29        producer.resume();
30      }
31      return item;
32  }}
```

Listing 1.3: Error prone Buffer implementation

The implementation of a value retrieval from the buffer – `removeItem` is complementary to the `putItem`. Firstly, there is a check if the buffer is empty, if so the suspension of the current thread (Consumer) is requested. If the buffer is not empty or the thread was resumed, the value is taken from the local buffer (the implementation was omitted), and the internal counter of buffer occupation is decremented. The last step is to resume execution of the suspended producer if the buffer was full, and now it contains a value that can be consumed.

There are several problems with that implementation, which stems from the race condition around access to the `itemCount`. One problem is of non-atomic operation in lines 15 and 25 that can lead to lost update. If Producer thread reads the value of `itemCount` and is preempted before storing the result back, the Consumer thread that will execute line 25 will load not yet updated value. The process will decrement it and store it to main memory. Then Producer thread can be resumed by the scheduler to increment the old value and store its value into main memory. Therefore, the size of the buffer will be inconsistent.

Another problem can occur while the buffer is full. At the same time, the producer thread is preempted after the execution of the test at the 11th line but before the execution of the 12th line. If then Consumer thread will drain the buffer from all its values it finally executes 27th line to resume execution of Producer thread. The next iteration of the loop will suspend the Consumer thread as there are no more values in the buffer. However, because the Producer thread was not yet properly suspended, the scheduler grants the CPU to that thread which will execute the 12th line to suspend itself. This situation results in deadlock as both threads were suspended. The resumption action of producer requested by the consumer is lost, and the system as a whole cannot progress.

An example of the Producer-Consumer problem solution with blocking buffer taking advantage of synchronisation primitives is presented in Listing 1.4.

```
class Buffer {
  final int BUFFER_SIZE = 3;
  volatile int itemCount = 0;

  boolean isFull() { return itemCount == BUFFER_SIZE;}
  boolean isEmpty() { return itemCount == 0; }

  synchronized void putItem(int item) {
    if (isFull()) {
      wait();
    }
    putItemIntoBuffer(item);
    itemCount = itemCount + 1;
    if (itemCount == 1) {
      notifyAll();
  }}

  synchronized int removeItem() {
    if (isEmpty()) {
      wait();
    }
    var item = removeItemFromBuffer();
    itemCount = itemCount − 1;
    if (itemCount == BUFFER_SIZE − 1) {
      notifyAll();
    }
    return item;
}}
```

Listing 1.4: Blocking buffer

1.5 Non-blocking algorithms

Another way to solve the concurrency synchronisation problem is through the usage of non-blocking algorithm. The field of research related to non-blocking algorithms is relatively more modern than concurrent computation based on classical synchronisation primitives. Non-blocking algorithms are a class of algorithms that can offer significantly higher scalability and liveness advantages. They can achieve it by coordination made at a finer level of granularity than lock-based algorithms. Another advantage is that they can greatly reduce scheduling overhead because they do not block execution of the thread when multiple threads compete for the same resource. Further, they can guarantee that their usage is deadlock-free and other liveness problems are avoided. However, they are considerably more complicated to design and implement than algorithms based on synchronisation primitives. Non-blocking algorithms achieve their goals based on the hardware supported atomic compare-and-swap – CAS operations. E.g. on Intel x86 microarchitecture, there are the following assembly instructions supported `CMPXCHG`, `CMPXCHG16B`, `CMPXCHG8B` that together with `LOCK` prefix are guaranteed to be atomic in multiprocessor environment [87, 88]. On the other hand, ARM processors are equipped with two instructions `LDREX` (load-link) and `STREX` (store conditionally) that have to be executed one after another [80].

A solution of the Producer-Consumer problem 1.4 belonging to the class of non-blocking algorithms was proposed as a lock-free queue [130]. This popular and simple implementation is used in production code, e.g. in Java Collection Framework as `ConcurrentLinkedQueue` [146, 107, 108]. In this case, however, the buffer is unbounded, that means that there is no limit imposed on the maximal number of elements to be stored. In Listing 1.5 an adapted version of the algorithm is presented. For brevity reason, only `putItem` method is presented.

```
class NonBlockingBuffer {
  boolean putItem(int value) {
    var newNode = new Node(value, null);
    while (true) {
      var t = tail;
      Node<E> next = t.getNext();
      if (t == tail) {
        if (next == null) {
          if (t.next.compareAndSet(next, newNode)) {
            tail.compareAndSet(t, newNode);
            return true;
          }
        } else {
          tail.compareAndSet(t, next);
}}}}
```

Listing 1.5: Non-Blocking buffer

It is not immediately evident that the algorithm works correctly. Moreover, that algorithm is not guaranteed to be starvation-free. The main advantage of that algorithm is that limitation of overhead related to the acquisition of the lock is removed and therefore, the performance of the queue is better [107].

One can distinguish a subclass of lock-free, non-blocking data structure: wait-free algorithms. Lock-free data structures guarantee overall system progress. The guarantee is achieved due to resigning from usage of locks and therefore avoiding all problems generated by the locks misusage (dead-locks, priority inversion). Wait-free data structures additionally guarantee the progress of every thread in the system, as they provide the non-starvation guarantee for concurrent algorithms. While practical lock-free implementations are known for various data structures, wait-free data structure designs are rare. However, the authors of the paper [176] present the algorithm that allows modification of already lock-free algorithm into the wait-free one with the cost of additional complexity added.

1.6 Safety and Liveness

In an ordinary sequential program, e.g. a program to sort a list of numbers, program correctness can be formulated in terms of preconditions and postconditions in a formalism such as Hoare's Logic because the program's underlying semantics can be viewed as given by a transformation from an initial state to a final state. Therefore, the testing approach occurred to be successful in the verification of such class of programs. Both precondition and postcondition can be controlled and specified in the code itself as a unit test. Therefore, track all the future modification of the underlying code.

However, for a concurrent class of the programs or those that have to work continuously interacting with environment, checking just preconditions and postconditions is not enough. All specific type of problems mentioned in the previous sections like deadlocks, live locks, starvation, priority inversion, race-conditions, and others are impossible to be verified with well-established testing methods. So the proving techniques to assure the quality of concurrent algorithms were developed. Initially, there were handwritten proves using a Floyd-Hoare style logic like, e.g. prove in non-blocking queue algorithm paper [130]. Probably, the best known formal system was the one proposed by Owicki and Gries [144] for reasoning about Conditional Critical Regions.

The problem of proving the correctness of concurrent algorithms can be approached by proving a list of requirements that always have to be correct for the given algorithm, apart from possible scheduling of the processes. Instead of happy–path operational thinking, one can focus on "what should be right?" as invariant-based reasoning avoids complexity and bugs of operational reasoning of concurrent systems.

The requirements can be grouped into two major groups of requirements, namely: safety and liveness properties. Informally, a *safety property* stipulates that "bad things" do not happen during execution of the program. So the safety property can be "there is no deadlock", "there is no race-condition". A *liveness property* stipulates that "good things" do happen (eventually) [99, 4]. Therefore, the liveness property can be "a thread will always be eventually executed" – there is no starvation. Distinguishing between safety and liveness properties is useful because proving that a program satisfies a safety property involves an invariance argument, while proving that a program satisfies a liveness property involves a well-foundedness argument. Thus, knowing whether a property is a safety or liveness helps in the decision of how to prove that the property holds.

Let S be the set of program states, S^{ω} the set of infinite sequences of program states, and S^{*} the set of finite sequences of program states. Here, only infinite execution of the program is considered. Formally, safety property could be defined as [3, 4]:

$$\forall \sigma \in S^{\omega} \colon \sigma \models \phi \iff \forall i \geq 0 \colon \forall \beta \in S^{\omega} \colon (\sigma[..i]\beta) \models \phi$$

So either ϕ does not hold for the system and there exist finite prefix of the state sequence of states where something "bad happen" or the opposite is true.

Liveness property could be defined as dual of the safety:

$$\forall \alpha \in S^{*} \colon \exists \beta \in S^{\omega} \colon \alpha\beta \models \phi$$

Many properties are neither safety nor liveness [3]. E.g., any property characterized as follows: *Eventually an event of type **E2** will happen and all preceding events are of type **E1**.* is the intersection of a safety property and a liveness property. The safety property is "no **E1** before **E2**; does not happen" and the liveness property is “**E2** eventually happens”. Every property ϕ is the intersection of a safety property and a liveness property [3].

1.7 Temporal Logics

Temporal logics [153] are modal logics [85] where the underlying frame structure is seen as a time structure, and where the modalities (called temporal operators or connectives) reflect this interpretation. Both modal and temporal logics have been used with great success in many fields of Computer Science.

In the late ’70s, Pnueli [152] and Owicki and Lamport [145] had proposed the use of Temporal Logics for specifying concurrent programs. Although they still advocated hand-constructed proofs, their work demonstrated convincingly that Temporal Logic was ideal for expressing concepts like mutual exclusion, absence of deadlock, and absence of starvation. The operators of temporal logic, such as sometimes and always, appear quite appropriate for describing the time-varying behaviour of such programs. Therefore, expressing safety and liveness requirements using temporal logic formalism is relatively easy. Additionally, for every temporal logic presented below, there exists efficient model checking algorithms that allow automatic, computer-aided verification of the constraints modelled in those logics [145].

Temporal logics has been used or proposed for use in virtually all aspects of concurrent program design, including specification, verification, manual program composition (i.e. development), and mechanical program synthesis.

There exist many Temporal Logic systems that could be classified along a number of axes [59]: propositional versus first-order, global versus compositional, branching versus linear, points versus intervals, and past versus future tense. However, only a few examples of logics will be presented in this chapter that could be characterised as global, point-based, discrete-time, future-tense logics. Both branching and linear time structure will be discussed [29].

Just before presenting a few temporal logics related to Alvis toolbox, a generic definition of the model is given in the form of a labelled transition system which will allow providing what it means that model satisfies a temporal logic formula. A labelled transition system can be defined as a 6-tuple:

$$LTS \coloneqq (S, A, R, S_0, AP, L) \tag{1.1}$$

where:

- S – is the set of states
- A – is the set of actions
- $R \subseteq S \times A \times S$ – is the transition relation
- $S_0 \subseteq S$ – is the set of initial states
- AP is the set of atomic propositions

- $L: S \to 2^{AP}$ is the labelling function

A path π is any infinite sequence of states that can be constructed from states in the transition system. $\pi = s_0 s_1 s_2 ...\ s_0 \in S_0$ and $\forall i: (s_i, a, s_{i+1}) \in R$. A trace of the path $trace(\pi)$ is an infinite word that is constructed by application of the labelling function to every state: $trace(\pi) = L(s_0)\, L(s_1)\, L(s_2) .. = \sigma_0\sigma_1\sigma_2 ...$ The set of all paths of the LTS is denoted by $Paths(LTS)$ and consists of all possible paths starting from every initial state of the LTS. The set of all traces of the LTS is denoted by $Traces(LTS)$ and consists of all the traces of the paths from the set $Paths(LTS)$

1.7.1 LTL

Linear Temporal Logic (shortened LTL) is a temporal logic introduced by Pnueli [152] in order to describe and verify reactive systems. The logic is defined for the linear paths. Temporal logic is used in various model checkers to describe system requirements that can be automatically verified. The minimal set of LTL expressions could be defined with the help of the following grammar productions:

$$\phi ::= a \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \mathsf{X}\,\phi \mid \phi_1\,\mathsf{U}\,\phi_2 \tag{1.2}$$

The operator X is the next operator and asserts the next state of the system, while U is the until operator that asserts that left-hand side expression has to be valid till the state where right-hand side expression is correct.

Formal semantics of the LTL formulae over the set of atomic propositions AP can be defined with the following definitions:

$$\begin{aligned}
\sigma \models a &\quad \text{iff} \quad a \in \sigma[0] \\
\sigma \models \phi_1 \wedge \phi_2 &\quad \text{iff} \quad \sigma \models \phi_1 \text{ and } \sigma \models \phi_2 \\
\sigma \models \neg\phi &\quad \text{iff} \quad \sigma \not\models \phi \\
\sigma \models \mathsf{X}\,\phi &\quad \text{iff} \quad \sigma[1..] \models \phi \\
\sigma \models \phi_1\,\mathsf{U}\,\phi_2 &\quad \text{iff} \quad \exists j \geq 0: \sigma[j..] \models \phi_2 \wedge \forall j > i \geq 0: \sigma[i..] \models \phi_1
\end{aligned}$$

By σ it is understood the inifinite word build over alphabet over AP set. $\sigma \in \left(2^{AP}\right)^{\omega}$. $\sigma[i]$ denotes the i-th letter of σ word. $\sigma[i..]$ denotes the infinite suffix of the σ by omission first $i - 1$ letters. Set of all possible words that model the formula ϕ is denoted by $Words(\phi)$.

Then we can say that the labelled transition system models the LTL formula ϕ when: $Traces(LTS) \subseteq Words(\phi)$.

One can extend the minimal set of the LTL connectives with the following formulae and operators and express them by abbreviation rules:

$$\begin{aligned}
true &\coloneqq a \vee \neg a \\
false &\coloneqq \neg true \\
\phi_1 \vee \phi_2 &\coloneqq \neg\left(\neg\phi_1 \wedge \neg\phi_2\right) \\
\phi_1 \implies \phi_2 &\coloneqq \neg\phi_1 \vee \phi_2 \\
\phi_1 \iff \phi_2 &\coloneqq \phi_1 \implies \phi_2 \wedge \phi_2 \implies \phi_1 \\
\mathsf{F}\ \phi &\coloneqq true\ \mathsf{U}\ \phi \\
\mathsf{G}\ \phi &\coloneqq \neg\mathsf{F}\ \neg\phi \\
\phi_1\ \mathsf{W}\ \phi_2 &\coloneqq \phi_1\ \mathsf{U}\ \phi_2 \vee \mathsf{G}\ \phi_1 \\
\phi_1\ \mathsf{R}\ \phi_2 &\coloneqq \phi_2\ \mathsf{W}\ \left(\phi_2 \wedge \phi_1\right)
\end{aligned}$$

The operator F is called in the future, or finally. It asserts that expression has to be valid in some state of the system. The operator G is called globally or always. It describes an expression that has to be valid in every state of the system. The operator W is a weak until operator. It is very similar to the until operator, but additionally, it can accept the path where right-hand side expression does not have to appear at all.

As examples of the LTL application in practise, one can take a look at the following generic system requirements. The complex requirements are accurately defined and therefore can be unambiguously model checked.

- reachability – $\mathsf{F}\ \phi$
- safety – $\mathsf{G}\ \neg\phi$
- liveness – $\mathsf{G}\ \mathsf{F}\ \phi$
- stability – $\mathsf{F}\ \mathsf{G}\ \phi$

1.7.2 CTL

Computation Tree Logic (shortened CTL) is a temporal logic introduced by Clarke and Emerson [40] in order to describe and verify reactive systems. In contrast to LTL, time structure forms a tree, and all the paths of the transition system can be taken into account by the single formula evaluation. At the same time, LTL analysed all the paths independently. As state may have more than one possible future, a new type of modal operators was introduced to distinguish if one intends to quantify all of the possible futures or existence of at least one path.

The minimal set of CTL expressions could be defined with the help of the following grammar production rules [122]:

$$\phi ::= a \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \mathsf{EX}\ \phi \mid \mathsf{E}\ \phi_1\ \mathsf{U}\ \phi_2 \mid \mathsf{EG}\ \phi \tag{1.3}$$

The operator E is an existence operator that assures that there is at least one path, starting from the considered state, for which some property holds. Path operators X , U , G in CTL, always have to be quantified by one of the operators over path either E, or its dual A – see below.

Formal semantics of the CTL formulae over the set of atomic propositions AP can be defined with the following definitions where $s \in S$:

$$\begin{aligned}
s \vDash a \quad & \text{iff} \quad a \in L(s)\\
s \vDash \phi_1 \wedge \phi_2 \quad & \text{iff} \quad s \vDash \phi_1 \text{ and } s \vDash \phi_2\\
s \vDash \neg\phi \quad & \text{iff} \quad s \nvDash \phi\\
s \vDash \mathsf{EX}\ \phi \quad & \text{iff} \quad \exists \pi \in Paths(s) : \pi[1] \vDash \phi\\
s \vDash \mathsf{E}\ \phi_1\ \mathsf{U}\ \phi_2 \quad & \text{iff} \quad \exists \pi \in Paths(s) : \exists j \geq 0 : \pi[j] \vDash \phi_2 \wedge \forall j > i \geq 0 : \pi[i] \vDash \phi_1\\
s \vDash \mathsf{EG}\ \phi \quad & \text{iff} \quad \exists \pi \in Paths(s) : \forall j \geq 0 : \pi[j] \vDash \phi
\end{aligned}$$

Then we can say that the labelled transition system models the CTL formula ϕ when: $\forall s \in S_0 : s \vDash \phi$.

One can extend the minimal set of the CTL connectives with the following operators and express them by the abbreviation rules:

$$\begin{aligned}
true &\coloneqq a \vee \neg a\\
false &\coloneqq \neg true\\
\phi_1 \vee \phi_2 &\coloneqq \neg(\neg\phi_1 \wedge \neg\phi_2)\\
\phi_1 \implies \phi_2 &\coloneqq \neg\phi_1 \vee \phi_2\\
\phi_1 \iff \phi_2 &\coloneqq \phi_1 \implies \phi_2 \wedge \phi_2 \implies \phi_1\\
\mathsf{EF}\ \phi &\coloneqq \mathsf{E}\ true\ \mathsf{U}\ \phi\\
\mathsf{AX}\ \phi &\coloneqq \neg\mathsf{EX}\ \neg\phi\\
\mathsf{A}\ \phi_1\ \mathsf{U}\ \phi_2 &\coloneqq \neg(\mathsf{E}(\neg\phi_2)\ \mathsf{U}\ \neg(\phi_1 \vee \phi_2) \vee \mathsf{EG}\ \neg\phi_2)\\
\mathsf{AG}\ \phi &\coloneqq \neg\mathsf{EF}\ \neg\phi\\
\mathsf{AF}\ \phi &\coloneqq \neg\mathsf{EG}\ \neg\phi
\end{aligned}$$

The operator A is all and assures that property holds for every path. The operator F , and G have the same meaning as in LTL.

LTL and CTL differ in expressive power. There are requirements expressible in LTL that cannot be translated into CTL, and another way round. As an example: F G a cannot be translated into CTL as AF AG a

is stricter requirement and $\mathsf{AF\ EG}\ a$ is weaker requirement than the LTL formula. However, an extensive comparison of the CTL and LTL logics in terms of expressive power, complexity, compositionality, and implementability can be found in the paper [182] and will not be discussed here further.

Both logics LTL and CTL can be used to pose requirements for a model in Alvis. Those logics are state-oriented as atomic propositions are boolean expressions asserting the state of the system. Alvis toolbox is able to create model in nuXmv model checker [21, 20, 34] that allows to verify formulae specified in LTL and CTL. Additionally, the master thesis by Kumoń [97] explored the possibility of implementation of LTL on-the-fly model checking algorithm within the Alvis toolbox.

1.7.3 μ-Calculus

μ-calculus is another branching-time logic. One of the versions of the logic is the propositional μ–calculus proposed by Kozen [96]. In that formulation, the logic is more expressive than CTL or LTL. However, the cost is hidden with time complexity of verification algorithm, which is exponential [60]. Therefore, practical applicability is infeasible.

As a remedy, there was suggested modification to the original formulation of the logic in the form of regular alternation-free μ-calculus [128]. The model-checking algorithm has linear-time complexity and was implemented in the EVALUATOR 3.0 model-checker within the CADP toolbox [61] using the generic OPEN/CAESAR environment for on-the-fly verification [66].

The regular alternation-free μ-calculus contains three types of formulae [128]: namely action formulae (denoted α), regular formulas (denoted β), and state formulas (denoted ϕ). The formulae can be expressed by the following grammar:

$$\alpha ::= a \mid \neg\alpha \mid \alpha \wedge \alpha \tag{1.4}$$

$$\beta ::= \alpha \mid \beta.\beta \mid \beta|\beta \mid \beta^* \tag{1.5}$$

$$\phi ::= true \mid false \mid \phi \vee \phi \mid \phi \wedge \phi \mid \langle\beta\rangle\phi \mid [\beta]\,\phi \mid Y \mid \mu Y.\phi \mid \nu Y.\phi \tag{1.6}$$

Semantics of the action formulae can be defined as follows:

$$\begin{aligned} Sat(a) &\coloneqq \{a\} \\ Sat(\neg\alpha) &\coloneqq Act \setminus \{a\} \\ Sat(\alpha_1 \wedge \alpha_2) &\coloneqq Sat(\alpha_1) \cup Sat(\alpha_2) \end{aligned}$$

Action formulae are built upon action names $a \in A$ and the standard boolean operators. The interpretation $Sat(alpha) \subseteq A$ of action formulae gives the set of LTS actions satisfying α. Derived boolean connectives are defined as before, e.g. $true \coloneqq a \vee \neg a$ for some a, $false \coloneqq \neg true$, $\alpha_1 \vee \alpha_2 \coloneqq \neg\,(\neg\alpha_1 \wedge \neg\alpha_2)$, etc.

Semantics of the regular formulae can be defined as follows:

$$\begin{aligned}
[\![\alpha]\!] &\coloneqq \left\{(s,s') \in S \times S \colon \exists a \in Sat(\alpha) : s \xrightarrow{a} s'\right\}\\
[\![\beta_1.\beta_2]\!] &\coloneqq [\![\beta_1]\!] \circ [\![\beta_2]\!]\\
[\![\beta_1|\beta_2]\!] &\coloneqq [\![\beta_1]\!] \cup [\![\beta_2]\!]\\
[\![\beta^*]\!] &\coloneqq [\![\beta]\!]^*
\end{aligned}$$

Regular formulae are built upon action formulae and the standard regular expression operators, namely concatenation :, choice |, and transitive-reflexive closure *. The interpretation $[\![\beta]\!] \subseteq S \times S$ of regular formulae gives a binary relation between the source and target states of transition sequences satisfying ($\circ$, $\cup$, and * denote respectively composition, union, and transitive–reflexive closure of binary relations). The empty sequence operator ϵ and the transitive closure operator $^+$ are defined as $\epsilon \coloneqq false^*$ and $\beta^+ \coloneqq \beta.\beta^*$.

Semantics of the state formulae can be defined as follows:

$$\begin{aligned}
Sat(true)_\rho &\coloneqq S\\
Sat(false)_\rho &\coloneqq \varnothing\\
Sat(\phi_1 \vee \phi_2)_\rho &\coloneqq Sat(\phi_1)_\rho \cup Sat(\phi_2)_\rho\\
Sat(\phi_1 \wedge \phi_2)_\rho &\coloneqq Sat(\phi_1)_\rho \cap Sat(\phi_2)_\rho\\
Sat(\langle\beta\rangle\,\phi)_\rho &\coloneqq \left\{s \in S \mid \exists s' \in S \colon (s,s') \in [\![\beta]\!] \wedge s' \in Sat(\phi)_\rho\right\}\\
Sat([\beta]\,\phi)_\rho &\coloneqq \left\{s \in S \mid \forall s' \in S \colon (s,s') \in [\![\beta]\!] \implies s' \in Sat(\phi)_\rho\right\}\\
Sat(Y)_\rho &\coloneqq \rho(Y)\\
Sat(\mu Y.\phi)_\rho &\coloneqq \bigcap\left\{S' \subseteq S \mid \Phi_\rho(S') \subseteq S'\right\}\\
Sat(\nu Y.\phi)_\rho &\coloneqq \bigcup\left\{S' \subseteq S \mid S' \subseteq \Phi_\rho(S')\right\}\\
&\quad \text{where } \Phi_\rho \colon 2^S \to 2^S, \Phi_\rho(S') = Sat(\phi)_{(\rho \oslash [S'=Y])}
\end{aligned}$$

State formulae are built upon propositional variables $Y \in \mathcal{Y}$ and the standard boolean operators, the possibility and necessity modal operators $\langle\beta\rangle\,\phi$ and $[\beta]\,\phi$, and the minimal and maximal fixed point operators $\mu Y.\phi$ and $\nu Y.\phi$. The μ and ν operators act as binders for Y variables in a way similar to quantifiers in first–order logic. A formula without free (not bound) occurrences of Y variables is denoted as closed. The interpretation $Sat(\phi) \subseteq S$ of state formulae, where the propositional context $\rho \colon \mathcal{Y} \to 2^S$ assigns state sets to propositional variables, gives the set of LTS states satisfying ϕ in the context of ρ (the $\oslash$ notation denotes context overriding: $(\rho_1 \oslash \rho_2)(Y)$ is equal to $\rho_2(Y)$ if Y is assigned by ρ_2 and to $\rho_1(Y)$ otherwise).

Here, *regular alternation–free* means that for every fixed point subformula $\mu Y.\phi$, the variable Y has no free occurrences inside a subformula of ϕ of the form $\nu Y.\phi'$ or $[\beta]\,\phi'$ where β contains a transitive-reflexive closure (a dual condition is imposed for maximal fixed point subformulae $\nu Y.\phi$).

The logic is action oriented and helps imposing assertion for the sequeces of actions of the system. Alvis toolbox is able to create model in CADP format and the actions can be model checked by CADP Evaluator [69, 170, 172].

1.8 Model Checking

The *Model Checking* problem could be stated as [39]:

Let M be a model of the system represented as a Labelled Transition System. Let ϕ be a formula of temporal logic (i.e., the specification). Find all states s of M such that $M, s \models \phi$.

In the previous sections, it was already shown what it means that a model satisfies the LTL, CTL, or μ–calculus formula. A *model checker* then is a tool intended for automatic, computer-based formal verification of the model. It takes as an input the model of a system from the above definition and a temporal logic property (or properties) and then explores the state space of the system in order to determine whether the model satisfies the property without a human interaction needed. If the property is true, then the model checker responds with success. If a property violation is encountered, then this is illustrated with a counterexample.

For all the logics introduced in this chapter, there exist efficient model checking algorithms that are implemented in various model checkers: SPIN [83], nuSMV/nuXmv [34, 30], PRISM [98, 138], TLA/TLC [104, 33], UPPALL [16, 17], CADP [69], to just name a few.

We can distinguish two classes of model checkers: an explicit state model checkers and symbolic model checkers. The difference is in the state space traversal mechanism used by both types of model checkers. An explicit state model checker builds an explicit representation of the state space together with all state transitions [84]. It does it by forwarding search from an initial state of the system and explores only reachable states. While symbolic model checker [26, 39] does not need to build the state space explicitly as reachability analysis can be performed based on symbolically encoded state, e.g. binary decision diagrams [25] which allows compressing states significantly. The search can happen in backward fashion from erroneous state and finding out if any initial state can be reached.

State explosion is a significant problem in model checking [39]. The number of global system states of a concurrent system with many processes or complicated data structures can be enormous. All model checkers suffer from this problem. The state explosion problem has been the driving force behind much of the research in model checking and the development of new model checkers.

Historically, the first works and papers on the explicit state model checkers were by Clarke and Emerson [40], and independently by Queille and Sifakis [154] appeared in early ’80s. However, in the paper [39], Clarke cites some earlier works on protocol verifications by Bochmann [22] and Holzmann from late ’70s. Though, those earlier version of Pan verifier (predecessor of SPIN) by Holzmann were not using temporal logics to describe system requirements.

Symbolic model checkers were introduced in the '90 [26] as a way to tackle the problem of the state space explosion. Symbolic model checkers are able to handle enormous state spaces of a number of states up to 10^{120} (in special cases) while explicit state model checkers can handle state spaces of 10^9 number of states [58]. However, symbolic model checkers perform backward exploration of the state space. As a result, they explore not only reachable states. Explicit state model checkers are more suitable for software verification and usually perform forward exploration of the state space. Therefore, only reachable states are explored. Symbolic model checkers are better suited for verification of electronic circuits and protocols. Providing a counter-example requires some additional effort from the symbolic model checker to compute the path.

There are several technics that allow limiting the state space of the problem [39, 84]. The most successful could be a partial order reduction. Partial order reduction techniques exploit the independence of concurrently executed events. Two events are independent of each other, when executing them in either order results in the same global state. Model Checking algorithms that incorporate the partial order reduction are described in several different papers, e.g. the *persistent sets* [70], the *stubborn sets* [179], and the *ample sets* [147]. They approach the problem differently, but contain many common ideas.

Another successful method can be *abstraction*. Abstraction is essential for reasoning about reactive systems that involve data. It is based on the observation that specifications of systems usually involve simple relationships among data values. For example, verifying a program may depend on simple arithmetical relationships. In such situations, abstraction can be used to reduce the complexity of Model Checking. The abstraction is usually specified by a mapping between data values in the system and a small set of abstract data values. By extending the mapping to states and transitions, it is possible to produce a much smaller, abstract version [42].

Last method mentioned here will be *Symmetry Reduction*. Finite-state concurrent systems often contain duplicated components, e.g., a system of identical communicating servers executing the same process. This information can be used to obtain reduced models. Having physical symmetry in a system often implies the existence of a non-trivial permutation group that preserves the transition graph. The permutation group can be used to define an equivalence relation on the state space. The resulting reduced model can be used to simplify the verification of temporal logic properties.

1.9 Alvis modelling language and Alvis Compiler

Alvis [166, 165, 163] is a formal modelling language designed primarily for development of systems consisting of concurrently operating units. It is actively being developed at AGH-University of Science and Technology in Kraków, Department of Applied Computer Science. The whole environment is intended to allow modelling and then finding and explaining the concurrency problems of the modelled system. The

Alvis language was developed based on many years of experience in the teaching of formal modelling using formalisms such as Petri nets [92], process algebras [19], and timed automata [18]. One of the goals taken into account while designing the Alvis formal modelling language was to provide a language with a simple syntax that is similar to the syntax of a general-purpose programming language. Thanks to that, creating a model is a relatively easy task. An additional requirement is to allow automatic model checking of the models. That way, experienced software engineers can easily and quickly achieve the right level of modelling skills. Contrary, in the case of the highly formalized and more abstract approaches mentioned in previous sections, it is much more challenging to accomplish the required modelling skills. The usage of a high-level programming language to specify dynamics of the system allows more straightforward translation between the production-used programming language and the model in Alvis. The translation could be both manual or done automatically by dedicated tools [129].

In the proposed approach, the heavy and not always easy to understand mathematical foundations are hidden from the user, without compromising the capabilities and expressive power of the formalism. That is achieved by the Alvis compiler tool that verifies the input: Alvis model described with high-level language and translates it into the mathematical model automatically. The current version of Alvis supports both timed [169, 174] and non-timed models [163]. The range of possible application starts from the modelling of concurrent, real-time, embedded systems to distributed multiserver computation systems [168, 173].

Alvis has its origins in the CCS process algebra [132, 1], the XCCS language [11, 164] and the Ada programming language [27]. The critical aspect that Alvis inherited from its predecessors is the concept of an agent, borrowed from CCS. An agent denotes any distinguished part of the system under consideration with defined identity persisting in time.

Alvis is equipped with a graphical language [165] for modelling communication channels between considered system units (called agents in Alvis) and a high-level programming language for defining agents' behaviour [172]. It is a formal language. Therefore, states of an Alvis model and transitions between them are defined unambiguously with mathematical rigour. The main traits of Alvis can be summarised as:

- possibility to verify and model check the formal model of the designed system;
- high-level language that defines the dynamics of the model components (agents);
- graphical language (graphical layer) that allows modelling the data and control flow;
- graphical hierarchisation of the model that allows to group agents into modules;
- system layer that allows switching between a different configuration of the system, e.g. multicore (so-called α^0) vs a single core (α^1).

The high-level language used for the description of the system dynamics is a combination of the Haskell functional programming language and C-like imperative language. Haskell is used for declaring data types of the variables and functions that operate on them. Usage of Haskell is limited to the pure (side effect free) functions. Therefore, it was possible to optimise arbitrarily complicated computations into a single state transition in a formal model. Imperative language is a formal part of the Alvis specification, and every statement of that language is formalised. The graphical layer allows specifying and visualise connections between agents and modularise the model with hierarchical diagrams. The last element of the Alvis description is the system layer that allows configuring number of available processing units (available CPUs or servers) or type of scheduling algorithm.

Before the detailed description of the Alvis language components will be provided in the next chapters, let us see how an example simplified model of blocking bounded buffer will look like in Alvis. In the Fig. 1.1 graphical layer of the designed system is presented. There are two types of agents active – Producer and Consumer representing threads or processes and a single passive agent – Buffer representing the shared resource. Every procedure of the passive agent in Alvis is equipped with a mutex that assures mutual exclusion in access to the passive agent and notification mechanism that once the procedure is available again notifies all awaiting agents.

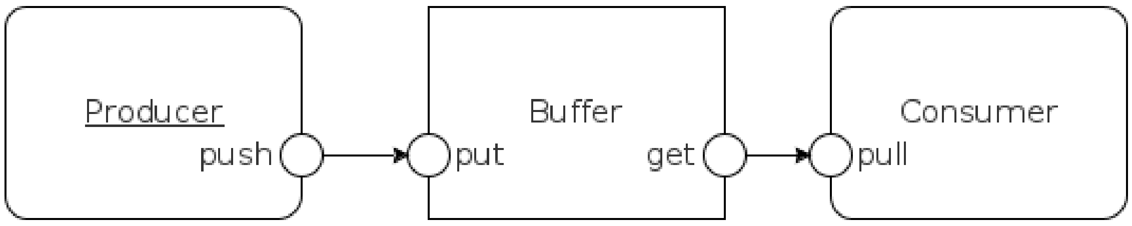


Figure 1.1: Graphical layer of Producer-Consumer problem in Alvis

In Listings 1.1 and 1.2, the behaviour of the agents – processes representing Producer and Consumer are presented.

```
agent Producer {
 value :: Int = 0;
 loop {
  value = pick [1..3];
  out push value;
 }
}
```

Listing 1.1: Producer Alvis model

The `Producer` performs actions in an infinite loop. The actions are a random elements selection (`pick`) from a list of 1, 2, and 3, and sending that value through the port `push`. In the graphical layer, we can verify that `push` port of `Producer` is connected to the `put` procedure of `Buffer` agent.

The `Consumer` is even simpler as it is only fetching the data from its `pull` port into its `value` variable in the infinite loop. In the graphical layer, we can verify that `pull` port of `Consumer` is connected to the `get` procedure of `Buffer` agent.

```
agent Consumer {
 value::Int = 0;
 loop {
  in pull value;
 }
}
```

Listing 1.2: Consumer Alvis model

In Listing 1.3, we can see a simplified design of the bounded-buffer problem using Alvis synchronisation primitives. It is simplified as buffer capacity is limited to just one element. In Appendix A, the version of the buffer with a maximum capacity of 3 elements is shown for comparison. The access to the local variables of the buffer is only possible through its procedures. Access to the passive agent procedure and evaluation of the guard statement is secured by mutual exclusion pattern. Therefore, there is no possibility of race conditions, and the model can be simplified drastically. There is no problem of coarse-grained locking as well as this is an only theoretical model and computation time is not relevant. In the case of the buffer, there are two procedures, both equipped with guard statements. Every procedure has to be finished with **`exit`** statement. The `put` procedure is available only when the buffer is not full. The first action is responsible for fetching the value from its `put` port and storing it into a local variable. Next, the status of the `isFull` variable is negated. The `get` procedure is available only when the buffer is full (it is not empty). The first action is sending the value through its port `get`. Next, the status of the variable `isFull` is negated.

```
agent Buffer {
 value :: Int = 0;
 isFull :: Bool = False;

 proc (isFull) get {
  out get value;
  isFull = not isFull;
  exit;
 }

 proc (not isFull) put {
  in put value;
  isFull = not isFull;
  exit;
 }
}
```

Listing 1.3: Simplified Blocking Bounded Buffer Alvis model

The model of bounded buffer is simplified by built-in mutexes into Alvis passive agents. The LTS graph for this model generated by execution of Alvis tool set is presented in Figure 1.2. The system after few steps stops as the agent `Consumer` was not enabled by default. It was a deliberate setting as to present various

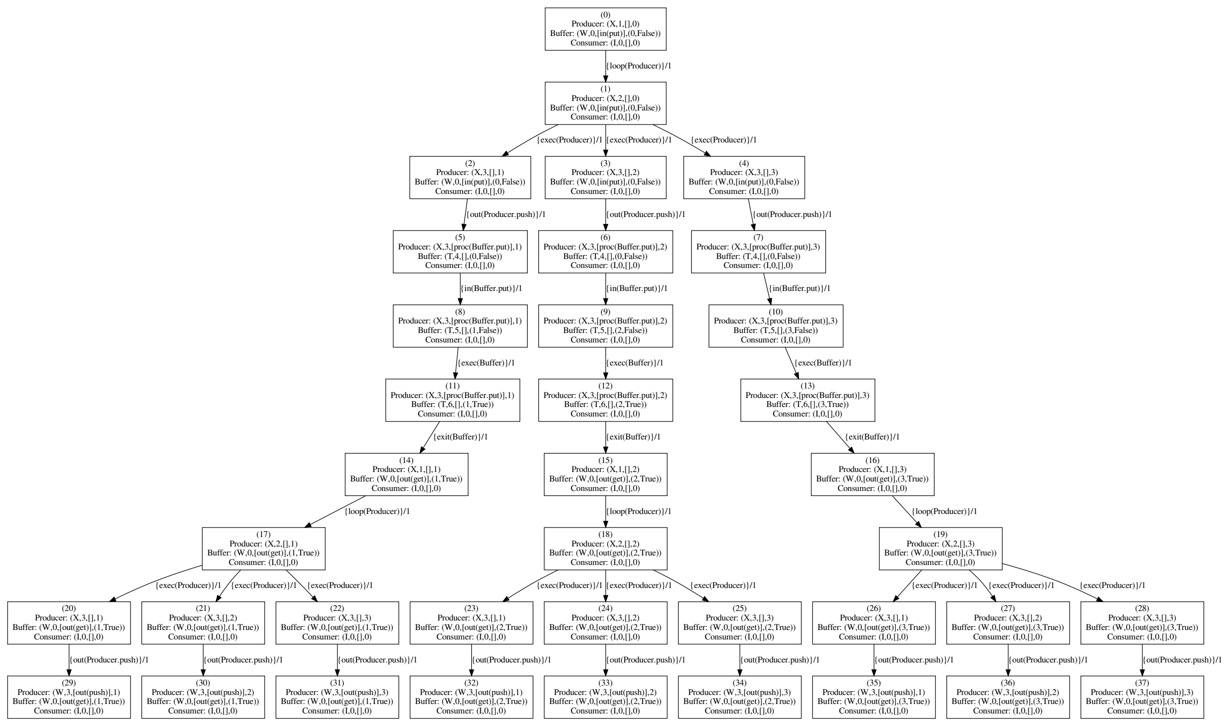

(0) Producer: (X,1,[],0) Buffer: (W,0,[in(put)],(0,False)) Consumer: (I,0,[],0)
{loop(Producer)}/1
(1) Producer: (X,2,[],0) Buffer: (W,0,[in(put)],(0,False)) Consumer: (I,0,[],0)
{exec(Producer)}/1
{exec(Producer)}/1
{exec(Producer)}/1
(2) Producer: (X,3,[],1) Buffer: (W,0,[in(put)],(0,False)) Consumer: (I,0,[],0)
(3) Producer: (X,3,[],2) Buffer: (W,0,[in(put)],(0,False)) Consumer: (I,0,[],0)
(4) Producer: (X,3,[],3) Buffer: (W,0,[in(put)],(0,False)) Consumer: (I,0,[],0)
{out(Producer.push)}/1
{out(Producer.push)}/1
{out(Producer.push)}/1
(5) Producer: (X,3,[proc(Buffer.put)],1) Buffer: (T,4,[],(0,False)) Consumer: (I,0,[],0)
(6) Producer: (X,3,[proc(Buffer.put)],2) Buffer: (T,4,[],(0,False)) Consumer: (I,0,[],0)
(7) Producer: (X,3,[proc(Buffer.put)],3) Buffer: (T,4,[],(0,False)) Consumer: (I,0,[],0)
{in(Buffer.put)}/1
{in(Buffer.put)}/1
{in(Buffer.put)}/1
(8) Producer: (X,3,[proc(Buffer.put)],1) Buffer: (T,5,[],(1,False)) Consumer: (I,0,[],0)
(9) Producer: (X,3,[proc(Buffer.put)],2) Buffer: (T,5,[],(2,False)) Consumer: (I,0,[],0)
(10) Producer: (X,3,[proc(Buffer.put)],3) Buffer: (T,5,[],(3,False)) Consumer: (I,0,[],0)
{exec(Buffer)}/1
{exec(Buffer)}/1
{exec(Buffer)}/1
(11) Producer: (X,3,[proc(Buffer.put)],1) Buffer: (T,6,[],(1,True)) Consumer: (I,0,[],0)
(12) Producer: (X,3,[proc(Buffer.put)],2) Buffer: (T,6,[],(2,True)) Consumer: (I,0,[],0)
(13) Producer: (X,3,[proc(Buffer.put)],3) Buffer: (T,6,[],(3,True)) Consumer: (I,0,[],0)
{exit(Buffer)}/1
{exit(Buffer)}/1
{exit(Buffer)}/1
(14) Producer: (X,1,[],1) Buffer: (W,0,[out(get)],(1,True)) Consumer: (I,0,[],0)
(15) Producer: (X,1,[],2) Buffer: (W,0,[out(get)],(2,True)) Consumer: (I,0,[],0)
(16) Producer: (X,1,[],3) Buffer: (W,0,[out(get)],(3,True)) Consumer: (I,0,[],0)
{loop(Producer)}/1
{loop(Producer)}/1
{loop(Producer)}/1
(17) Producer: (X,2,[],1) Buffer: (W,0,[out(get)],(1,True)) Consumer: (I,0,[],0)
(18) Producer: (X,2,[],2) Buffer: (W,0,[out(get)],(2,True)) Consumer: (I,0,[],0)
(19) Producer: (X,2,[],3) Buffer: (W,0,[out(get)],(3,True)) Consumer: (I,0,[],0)
{exec(Producer)}/1
{exec(Producer)}/1
{exec(Producer)}/1
{exec(Producer)}/1
{exec(Producer)}/1
{exec(Producer)}/1
{exec(Producer)}/1
{exec(Producer)}/1
{exec(Producer)}/1
(20) Producer: (X,3,[],1) Buffer: (W,0,[out(get)],(1,True)) Consumer: (I,0,[],0)
(21) Producer: (X,3,[],2) Buffer: (W,0,[out(get)],(1,True)) Consumer: (I,0,[],0)
(22) Producer: (X,3,[],3) Buffer: (W,0,[out(get)],(1,True)) Consumer: (I,0,[],0)
(23) Producer: (X,3,[],1) Buffer: (W,0,[out(get)],(2,True)) Consumer: (I,0,[],0)
(24) Producer: (X,3,[],2) Buffer: (W,0,[out(get)],(2,True)) Consumer: (I,0,[],0)
(25) Producer: (X,3,[],3) Buffer: (W,0,[out(get)],(2,True)) Consumer: (I,0,[],0)
(26) Producer: (X,3,[],1) Buffer: (W,0,[out(get)],(3,True)) Consumer: (I,0,[],0)
(27) Producer: (X,3,[],2) Buffer: (W,0,[out(get)],(3,True)) Consumer: (I,0,[],0)
(28) Producer: (X,3,[],3) Buffer: (W,0,[out(get)],(3,True)) Consumer: (I,0,[],0)
{out(Producer.push)}/1
{out(Producer.push)}/1
{out(Producer.push)}/1
{out(Producer.push)}/1
{out(Producer.push)}/1
{out(Producer.push)}/1
{out(Producer.push)}/1
{out(Producer.push)}/1
{out(Producer.push)}/1
(29) Producer: (W,3,[out(push)],1) Buffer: (W,0,[out(get)],(1,True)) Consumer: (I,0,[],0)
(30) Producer: (W,3,[out(push)],2) Buffer: (W,0,[out(get)],(1,True)) Consumer: (I,0,[],0)
(31) Producer: (W,3,[out(push)],3) Buffer: (W,0,[out(get)],(1,True)) Consumer: (I,0,[],0)
(32) Producer: (W,3,[out(push)],1) Buffer: (W,0,[out(get)],(2,True)) Consumer: (I,0,[],0)
(33) Producer: (W,3,[out(push)],2) Buffer: (W,0,[out(get)],(2,True)) Consumer: (I,0,[],0)
(34) Producer: (W,3,[out(push)],3) Buffer: (W,0,[out(get)],(2,True)) Consumer: (I,0,[],0)
(35) Producer: (W,3,[out(push)],1) Buffer: (W,0,[out(get)],(3,True)) Consumer: (I,0,[],0)
(36) Producer: (W,3,[out(push)],2) Buffer: (W,0,[out(get)],(3,True)) Consumer: (I,0,[],0)
(37) Producer: (W,3,[out(push)],3) Buffer: (W,0,[out(get)],(3,True)) Consumer: (I,0,[],0)

Figure 1.2: Labelled Transition System of Producer Consumer problem

configuration of agents in a simple model. In real model, both agents would be configured in the running mode or one of the agents could initiate execution of `Consumer`.

The Alvis model for non-blocking algorithm can be found in the Appendix A.

1.10 Thesis statement

The primary goal of the thesis is to provide a tool that allows automated generation of Labelled Transition System from Alvis formal model for the untimed Alvis language version with α^0 system layer.

To achieve that goal, the following intermediate goals can be specified:

- Design a set of algorithms that will transform the formal Alvis model into its Intermediate Haskell Representation.
- Implementation of the proposed algorithms in the form of the Alvis compiler – tool that automatically verifies syntactical correctness of the Alvis model design and is able to create an Intermediate Haskell Representation reflecting the designed model.
- The last goal is to provide an efficient algorithm as part of the Intermediate Haskell Representation to create a Labelled Transition System.

1.11 Overview of contributions

First, this dissertation leads the reader through the brief introduction to the Alvis formal modelling language. It starts forming the definition of modelling language components, so-called layers of the model. The semantics and meaning of the Alvis instruction set is provided, together with the definition of the system state and possible transitions between states.

Later on, the main contribution of the author of this dissertation is presented. Significant work was about to design and implement computationally efficient algorithms that allow transforming the initial Alvis model into a program – Intermediate Haskell Representation.

Secondly, the design, implementation, and performance measurement of the algorithm that allows to efficiently explicit model of the system in the form of Labelled Transition System is described.

The content of the dissertation is as follows:

- Chapter 2 addresses other formal languages that could be compared to Alvis toolchain. The comparison is accompanied by example models to better visualise differences between Alvis and discussed formalisms.
- Chapter 3 presents an introduction to Alvis formal modelling language. The topics contained within this chapter, i.e. elements of Alvis language like: the formal definition of modelling layers, instruction semantics, and rules governing the behaviour of the model. Those are just a short presentation of the former work by professor M. Szpyrka [163, 169]. However, since then a few modifications of Alvis instruction set were introduced like: non-blocking communication instructions, removal of the instruction `if` (as redundant to the instruction **`select`**), simplification of passive agent instructions semantics by requiring every procedure to end with the explicit instruction **`exit`** were later added based on the feedback from progress on the Alvis compiler implementation. In that chapter, the updated version is presented. The elements of the Alvis formal modelling language that are essential to understand later parts of the dissertation are described.
- Chapter 4 presents the main contribution of the dissertation. There are discussed algorithms and data structures that allow transforming model designed in Alvis language into Intermediate Haskell Representation. The algorithms are an efficient implementation of the formal rules introduced in the former chapter. Intermediate Haskell Representation is the source code of the program that allows Labelled Transition System generation in various formats. The algorithms presented there were designed and later on implemented in the Alvis Compiler tool by the author of the dissertation.
- Chapter 5 contains discussion of the approach to design and implement of the algorithm that is responsible for the construction of the Labelled Transition System, which is included into Intermediate Haskell Representation source code.

- Chapter 6 discusses performance measurement of the graph exploration algorithm. The performance of the algorithm is of paramount importance, as a more efficient algorithm allows processing more complex and advanced systems.
- Chapter 7 contains a summary of the dissertation. Apart from walkthrough discussion of the main achievements presented in the dissertation, the future plans and possible extensions of the Alvis toolchain and language are discussed.

Chapter 2

Related work

2.1 Petri Nets

Petri Nets is one of the most popular formal modelling formalism. They allow for graphical specification and formal verification of concurrent systems. The name comes from the author Carl Petri [149] who firstly introduced the formalism in the '60s. Since then, many variants, classes and extensions were proposed. In 2019, 40th International Conference on Application and Theory of Petri Nets and Concurrency co-located with ACSD took place [55].

A Petri net is a directed bipartite graph that has two types of vertices, namely: places and transitions. Places are depicted as circles and transitions are depicted as rectangles. An edge can only connect a place to a transition or other way round. A place can contain any number of tokens, depicted as black circles. Configuration of a net when states contain a concrete number of tokens is called *marking*. A transition is *enabled* if all places connected by an incoming edge contain at least one token. Then the transition can *fire*. Fire is related to tokens consumption from incoming places and tokens production in outgoing places. In Fig. 2.1 a model for the Producer-Consumer problem is shown. The net marking is that the Producer is in idle state, and the Consumer is in waiting state. The Producer is ready to produce a value as its transition *produce* is enabled.

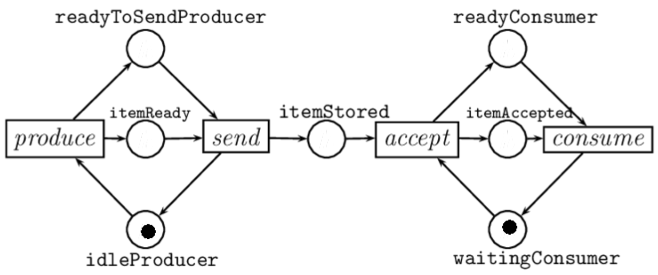


Figure 2.1: Petri Net Producer-Consumer problem. Adapted from Velardo [156]

The main difference between Petri Nets formalism and Alvis language is a steeper learning curve for mastering Petri Nets, especially for an experienced software engineer. Creation of an even simple model requires lengthy training, as Petri Nets concept does not relate straightforward to programming languages syntax. Additionally, more complicated models become quickly too big to be easily maintainable and easily understood. One of the most similar Petri Net class to Alvis language could be Timed Coloured Petri Nets (Timed CPNs) [91]. It is an extension of standard Petri Net where there are several types of tokens that differ by their colours. Additionally, tokens have time marker. The colour of the token corresponds to the type of variable in the programming language. CPN's colouring scheme can be defined in the functional programming language: Standard ML [133] together with expressions that simplify data manipulation [134]. As time markers can grow into infinity, reachability graphs for that type of Petri Net may be infinite. That impacts standard verification techniques for Petri Nets and requires careful treatment and specialised verification approach. In Alvis, there is no global clock, and only relative time is considered while computing the LTS. Therefore the LTS graph does not grow infinitely due to time-lapse. Another difference is that the description of the Petri Net system is low level and therefore hardly resembles a modelled system. In contrast, Alvis active agents can be easily matched with processes or threads of the modelled system. Last difference that can be mentioned is the expressivity of the underlying type system. Alvis allows an arbitrary type to be used in the model description. New user-defined types and functions to manipulate them can be added while SML allows an only limited set of types and operators.

2.2 Process Algebras

Another popular group of formalisms to model distributed systems is process algebras. The system behaviour is modelled as a sequence of algebraic expressions. One of the most common process algebra is Calculus of Communicating Systems, CCS [132, 1], Communicating Sequential Processes [78, 19], and LOTOS [24].

Modelling of the system requires providing a specification script with set of algebraic equations (CCS, CSP) or code like description (LOTOS). In the case of CCS and CSP communication between process is specified implicitly if two processes have action with the same name it is the synchronisation point. In the case of more complex systems, it poses a risk of creation of an accidental communication channel by sharing the same action name just by accident. LOTOS is standardised by ISO community as a specification language for communication protocols. In LOTOS, processes communicate synchronously through explicitly defined gateways. The description language at a glance looks like a high-level programming language, but in reality, it is an algebraic language inspired by CCS and CSP.

The LOTOS equations are expected to satisfy the termination property and the implementation of Inria CADP pose limits on using recursive processes in the specification. The example of Producer-Consumer problem is presented in Listing 2.1.

```
specification Producer_Consumer [ pc, cc ] : exit

behavior (
  Producer [ pc ]
  |||
  Consumer [ cc ])
  ||
  Channel [pc, cc]

where
  process Producer [ out ] : noexit :=
    out;
    Producer [out]
  endprocess
  process Consumer [ in ] : noexit :=
    in;
    Consumer [in]
  endprocess
  process Channel [ le, re ] : noexit :=
    le;
    re;
    Channel [ le, re ]
  endprocess
endspec
```

Listing 2.1: LOTOS Producer-Consumer problem. Adapted from Velardo [117]

2.3 nuXmv

NuSMV [35, 34], and newer NuXmv [30] is a modelling language that allows simulation and performing symbolic model checking. It was developed in cooperation between teams at Carnegie Mellon University, University of Trento and Fondazione Bruno Kessler (FBK). The first version appeared in 1999, and it is still maintained.

Modelling language allows to specify the system as a composition of modules and creates a state machine as a synchronous execution of the particular modules. The asynchronous composition is no longer supported. Finite-state systems can be verified through state-of-the-art SAT-based model checking algorithms while infinite-state systems (e.g. systems with real and integer variables) through SMT-based techniques running on top of MathSAT5 [36].

The module may use variables of a limited number of types like boolean, enumeration, integer, or real, and arrays. Evolution of the module can be described with *ASSIGN* declarations that specify how variables should change between states, or *TRANS* constraints that are boolean equations that allow posing conditions on current and next state variables. The modeller may additionally add fairness assumptions to eliminate trivial counter-examples. Synchronous execution is beneficial for modelling electronic circuits. However, modelling software requires asynchronous modelling as threads may be interleaved and scheduled in different orders, leading to bugs. Limiting the execution to just synchronous scheduling makes sense for electronic circuits, but for software it hides race conditions and therefore some bugs in software may be not possible to find.

Requirements may be defined in LTL, CTL, and RTCTL temporal logics. RTCTL is a variant of CTL with added temporal operators that can be expanded with the original CTL operators. The example of Producer-Consumer problem is presented in Listing 2.2.

```
MODULE Buffer
VAR
  isFull : boolean;

MODULE Producer(buffer)
VAR
  s : {idle, itemReady}
ASSIGN
  next(s) := case
    s = idle & !buffer.isFull : itemReady;
    TRUE : s;
  esac;
TRANS
  !buffer.isFull & next(s) = itemReady –> next(buffer.isFull) = true

MODULE Consumer(buffer)
VAR
  s : {wait, itemAccepted}
ASSIGN
  next(s) := case
    s = wait & buffer.isFull : itemAccepted;
    TRUE : s;
  esac;
TRANS
  buffer.isFull & next(s) = itemAccepted –> next(buffer.isFull) = false

MODULE main
VAR
  buffer : Buffer;
  producer : Producer(buffer);
  consumer : Consumer(buffer);
```

Listing 2.2: NuXmv Producer-Consumer problem.

2.4 CADP

Construction and Analysis of Distributed Processes (CADP) [69] is a set of various tools for modelling, simulation, and verification of asynchronous concurrent systems such as communication protocols, distributed systems, web services, or multiprocessor programs. It is supported by CONVECS research team at Inria Grenoble – Rhône-Alpes and the LIG laboratory. First projects CAESAR and ALDEBARAN were published in the mid-'80s. First one is a compiler supporting LOTOS language 2.2, the later project allows processing labelled transitions systems, e.g. minimisation of the system according to some equivalence relation. Now, the toolset offers 42 different tools and more than 100 integration with other scientific tools. Tools support a complete process of designing the asynchronous system: creation of specification, simulation, prototyping, verification, testing and performance analysis. The package supports several modelling languages:

- LOTOS – already discussed in 2.2.
- Finite State Process (FSP) [119] – another algebraic notation for concurrent systems supported by LTSA tool.
- EXP [105] – a language to describe communication between systems modelled as labelled transition systems in the format Binary Coded Graphs (BCG).
- LOTOS NT [32] – refreshed version of LOTOS with a unified approach to define behaviour and data in a single system description instead of two different languages of the specification.

Additionally, the toolset provides two model checkers:

- eXecutable Temporal Language (XTL) [67] – functional language supporting exploration and querying transition systems defined in BCG format. Queries may include temporal logic formulae and some additional properties based on the state or the transition counting, or other recursive functions.
- Evaluator – a family of model checkers that accept requirements specified in alternation-free regular μ calculus and a system defined as a labelled transition system. Currently, there are three versions of the model checker: Evaluator3 [128, 68], Evaluator4 [69] allows specifying fairness assumptions, and support richer transition labels with a description of modified variables, and Evaluator5 [127] that supports probabilistic operators and Probabilistic Transition System (PTS).

The Evaluator model checker allows verification of an action oriented system where states are indistinguishable. That optimisation stems from communication systems, where the internal state is usually not observable. Thanks to that, it allows model checking of more complex systems up to 10^{10} states.

2.5 Uppaal

UPPAAL is a set of several tools for modelling, simulation and verification of real-time systems [16]. It was developed as a cooperation between Centre of Basic Research in Computer Science at Aalborg University in Denmark and Department of Information Technology at Uppsala University in Sweden. The first version was released in 1995 [17], since then it is still developed and maintained. The tool is equipped with its proprietary modelling language, simulator and dedicated symbolic model checker. Modelling language allows a system defined as a network of parallel time automata extended with clocks and variables. Variables can be manipulated in a similar fashion as in programming languages. A state of the model is defined as a vector of states of all automatas, their clocks, and valuation of all variables. Any automaton may trigger its transition independently or in synchronisation with other automaton [16]. The requirements can be specified in a simplified version of Timed CTL logic [5] that is an extension of CTL with temporal operators that include time intervals information. The example of Producer-Consumer problem is presetend in Listing 2.3 [16].

```
// Declarations
bool isFull = false;
int bufferValue = 0;

// Processes
process Producer {
  state idle, itemReady;
  init idle;

  trans idle -> itemReady {
    guard isFull == false;
    assign bufferValue := 3, isFull := true;
  },

  itemReady -> idle {
    guard true;
  };
}

process Consumer {
  state wait, itemAccepted;
  init wait;

  trans wait -> itemAccepted {
    guard isFull == true;
    assign bufferValue := 0, isFull := false;
  },

  itemAccepted -> wait {
    guard true;
  };
}

// System Configuration
system Producer, Consumer;
```

Listing 2.3: UPPALL Producer-Consumer problem

2.6 PRISM

PRISM [98] is a probabilistic symbolic model checker that allows analysis of several probabilistic models including: probabilistic automata – PA, various Markov models, and probabilistic timed automata – PTA. The last-mentioned model incorporates the notion of time [138]. PRISM accepts the specification of the model within a textual file. It extends the concept of state machine, adding the probabilistic transitions, clock variables, modularisation of the model into modules. The model description incorporates bounded variables limited to only several built-in types (boolean or finite range of integers) and a minimal number of built-in functions. On the other hand, Alvis allows arbitrary function and data types in its models because its expression is from general-purpose programming language – Haskell. The communication between modules in PRISM is explicitly declared within the particular entity specification by explicit usage of variables from different modules which can lead to hard to spot errors. Alvis interactions between models are visualised in its graphical layer. The model description requires from an engineer thinking about the states and the passing time to model the system properly. In that sense, the description of the PRISM system is a low level (resembles more the Alvis LTS graph) compared to higher abstraction provided by the Alvis language. However, the main advantage of PRISM is its probability-centric description of the model. Uncertainty modelling in Alvis is limited to several instructions: `pick` which returns a single element from the given list with a uniform and not adjustable probability and unblocking input or output communication that can fail or succeed while being scheduled and executed due to unavailability of other parts of the system. As a quick comparison of the languages, one can compare a failing communication model in Listings 2.4 and 2.5. A model in PRISM consists of several interacting modules. Each module consists of several finite-valued variables corresponding to the module state, and some guarded commands defining the transitions of a module of the form:

```
[<action>] <guard> -> <prob_1> : <update_1>
   + ...
   + <prob_n> : <update_n>;
```

A command consists of an optional action label, guard expression and probabilistic choice between updates. A guard is a predicate over variables, while an update specifies, using primed variables, how the variables of the module are updated when the command is executed. Interaction between modules is through guards (as guards can refer to variables from other modules) and action labels that allow modules to synchronise.

Listing 2.4: PRISM model of failing communication

```
pta
const int N;
module transmitter
  // Local variables
  s : [0..3] init 0;
  tries : [0..N+1] init 0;
  x : clock;
  invariant
    (s=0 => x<=2) & (s=1 => x <=5)
  endinvariant
  // Guarded commands
  [send] s=0 & x >= 1 & tries <= N –>
          0.9 : (s’ = 3)
      + 0.1 : (s’ = 1) & (tries’=tries’+1) & (x’=0);
  [retry] s=1 & x >= 3 –> (s’=0)&(x’=0);
  [quit] s=0 & tries > N –> (s’=2);
endmodule
rewards "energy" (s=0) : 2.5; endrewards
```

Listing 2.5: Code layer of the corresponding Alvis model

```
agent Transmitter {
  tries :: Int = 0;
  N :: Int = 0;

  in init N;

  loop (every 5) {
    out (2) send {
      success {
        jump end;
      }
      fail {
        tries = tries + 1;
        select {
          alt (tries >= N) {
            jump end;
  }}}}}
end:
  null;
}
```

2.7 TLA+

TLA+ [100, 76, 33] is a formal specification modelling language for verifying distributed and concurrent systems. It was created by Leslie Lamport, and the first version was released in 1999. It is worth to mention that Leslie Lamport was awarded Turing award for his contribution to the theory and practice of distributed and concurrent systems, notably the invention of concepts such as causality and logical clocks, safety and liveness, replicated state machines, and sequential consistency.

Systems are modelled as state machines, consisting of a set of variables; a finite set of assignments so-called "actions". Assignments may contain various math formulas. Many concepts, like natural number or elements of set theory, are defined in modules. The syntax of TLA+ is expressive but not that easy to understand by a software engineer. However, PlusCal was introduced in 2009 [102] to provide pseudocode like syntax and algorithms specified in it can be translated conditionally into TLA+ and model checked. An interesting feature of TLA+ is that their specs can be attractively rendered in LaTeX which should not be surprising taking into account that Lamport is the creator of LaTeX. TLC is a model checker for TLA+ [104].

An example model of Producer-Consumer problem in TLA+ is presented in Listing 2.6 and dining philosophers problem in PlusCal in Listing 2.7.

TLA+ has some successful production-grade use cases, e.g. Amazon Web Services has used TLA+ since 2011. TLA+ model checking uncovered bugs in DynamoDB, S3, EBS, and an internally distributed lock manager [136, 6]; Microsoft Azure used TLA+ to design Cosmos DB, a globally-distributed database with five different consistency models [131].

Listing 2.6: Producer-Consumer problem in TLA+ [101].

```
---------------------------- MODULE InnerFIFO --------------------------------
EXTENDS Naturals, Sequences
CONSTANT Message
VARIABLES in, out, q
InChan == INSTANCE Channel WITH Data <- Message, chan <- in
OutChan == INSTANCE Channel WITH Data <- Message, chan <- out
-----------------------------------------------------------------------------
Init == /\ InChan!Init
        /\ OutChan!Init
        /\ q = << >>

TypeInvariant == /\ InChan!TypeInvariant
                 /\ OutChan!TypeInvariant
                 /\ q \in Seq(Message)

SSend(msg) == /\ InChan!Send(msg) \* Send msg on channel 'in'.
              /\ UNCHANGED <<out, q>>

BufRcv == /\ InChan!Rcv \* Receive message from channel 'in'.
```

```
             /\ q' = Append(q, in.val) \* and append it to tail of q.
             /\ UNCHANGED out

BufSend == /\ q # << >> \* Enabled only if q is nonempty.
              /\ OutChan!Send(Head(q)) \* Send Head(q) on channel 'out'
              /\ q' = Tail(q) \* and remove it from q.
              /\ UNCHANGED in

RRcv == /\ OutChan!Rcv \* Receive message from channel 'out'.
        /\ UNCHANGED <<in, q>>

Next == \/ \E msg \in Message : SSend(msg)
        \/ BufRcv
        \/ BufSend
        \/ RRcv

Spec == Init /\ [][Next]_<<in, out, q>>
-----------------------------------------------------------------------------
THEOREM Spec => []TypeInvariant
=============================================================================
```

Listing 2.7: Dining philosophers problem modelled in TLA+ with PlusCal extension

```
EXTENDS Integers, Sequences, TLC, FiniteSets
CONSTANTS NumPhilosophers, NULL
ASSUME NumPhilosophers > 0
NP == NumPhilosophers

(* --algorithm dining_philosophers

variables forks = [fork \in 1..NP |-> NULL]

define
LeftFork(p) == p
RightFork(p) == IF p = NP THEN 1 ELSE p + 1

HeldForks(p) ==
  { x \in {LeftFork(p), RightFork(p)}: forks[x] = p}

AvailableForks(p) ==
  { x \in {LeftFork(p), RightFork(p)}: forks[x] = NULL}

end define;
process philosopher \in 1..NP
variables hungry = TRUE;
begin P:
  while hungry do
    with fork \in AvailableForks(self) do
      forks[fork] := self;
```

```
        end with;
        Eat:
          if Cardinality(HeldForks(self)) = 2 then
            hungry := FALSE;
            forks[LeftFork(self)] := NULL ||
            forks[RightFork(self)] := NULL;
          end if;
    end while;
  end process;
  end algorithm; ∗)
```

2.8 SPIN

SPIN [83, 82] is a general tool for verifying the correctness of concurrent software models in a rigorous and mostly automated fashion. It was written by Gerard J. Holzmann and others in the original Unix group of the Computing Sciences Research Center at Bell Labs, beginning in 1980. The software has become available for the public since 1991, and to this day it is still maintained, developed and continues to evolve. It was one of the earliest model checkers to be developed. However, the possibility of specifying temporal logic requirements was added later in the late 90s [39]. Moreover, it should be classified as an explicit-state model checker.

Systems to be verified are described in Promela (Process Meta Language), which supports modelling of asynchronous distributed algorithms as non-deterministic automata (SPIN stands for "Simple Promela Interprete"). Properties to be verified are expressed as Linear Temporal Logic (LTL) formulae. Model-checking of LTL formulae takes a standard procedure of their negation and conversion into Büchi automata. Then paths with cycles extracted from the transition system created from the model are checked by advancing Büchi automata for every state. If the process finishes with the accepting state, it means that the requirement violation was encountered and the path is a counter-example. Otherwise, the system is correct.

The main interest area of the model checker is verification of concurrent software rather than electronic circuits. So it is used when verifying distributed systems such as operating system, communication protocols, or concurrent algorithms.

Promela, the specification language of the tool is an imperative language similar to C. Additionally, the language was extended with primitives that allow defining concurrent processes, synchronisation primitives and communication channels. The expression of Promela can be labelled, and the requirement can refer to those labels. SPIN does not perform model-checking itself, but instead generates C sources for a problem-specific model checker. According to authors, that technique saves memory and improves performance, while also allows the direct insertion of chunks of C code into the model after generation was done.

In addition to model-checking, SPIN can also operate as a simulator, following one possible execution path through the system and presenting the resulting execution trace to the user. SPIN also offers numerous options to speed up the model-checking process further and save memory, such as [82]:

- partial order reduction;
- state compression;
- bitstate hashing (instead of storing whole states, only their hash code is remembered in a bitfield; this saves a lot of memory but voids completeness);
- weak fairness enforcement.

SPIN allows as well to perform on-the-fly model checking. This approach allows finishing the model checking task earlier if the bug is present in the system and does not require full state space exploration to start and perform model checking. This may save some time during the design phase to quickly receive feedback if bugs are encountered. To allow the on-the-fly model checking, a subset of LTL without X operator was selected [43] to specify the model requirements.

```
mtype = {P,C};
mtype turn = P;
chan buffer = [1] of {byte};

active proctype Producer() {
  do
  :: (turn == P) ->
        printf("Produce\n");
        turn = C
        buffer!42
  od
}

active proctype Consumer() {
  do
  :: (turn == C) ->
        printf("Consume\n");
        turn = P
        buffer?x
  od
}
```

Listing 2.8: SPIN Producer-Consumer problem. Adapted from Trentin [177]

2.9 Other formalisms

There are many other tools for formal modelling that allow the model to be verified. Below is mentioned the non-exhaustive list of other formalisms:

- *Java Path Finder* [75, 23] – an implementation of the Java Virtual Machine that allows to take control over scheduler and instruct it. Therefore, the Java code can be directly verified if it contains race-conditions, deadlocks and other concurrency problems. However, the tool is not actively developed and currently only supports an old version of Java 8. The software supports only a few core Java libraries. Due to lack of any abstraction checking, even medium code size program can be not feasible in practice.

- *MoonWalker* – inspired by Java Path Finder a tool for .NET environment.
- *Dafny* [113, 110, 111, 112] – Dafny is an imperative compiled language that targets C# and supports formal specification through preconditions, postconditions, loop invariants and loop variants. It supports proofs of imperative programs based on the Hoare logic. There is a possibility to build proves of concurrent programs.
- *Armada* [118] – an imperative C-like language for verification of high-performance concurrent programs. Verification is based on an SMT solver.
- *Coq* [14] – Coq is a formal proof management system. It provides a formal language to write mathematical definitions, executable algorithms and theorems together with an environment for semi-interactive development of machine-checked proofs. The work on Coq was initiated in 1984 from an implementation of the Calculus of Constructions at INRIA-Rocquencourt by Thierry Coquand and Gérard Huet.
- *Agda* [137] – Agda is a dependently typed functional programming language developed as an extension of Haskell. Agda is also a proof assistant like Coq. It is an interactive system for writing and checking proofs. However, it is missing documentation, which render it harder to learn.

Chapter 3

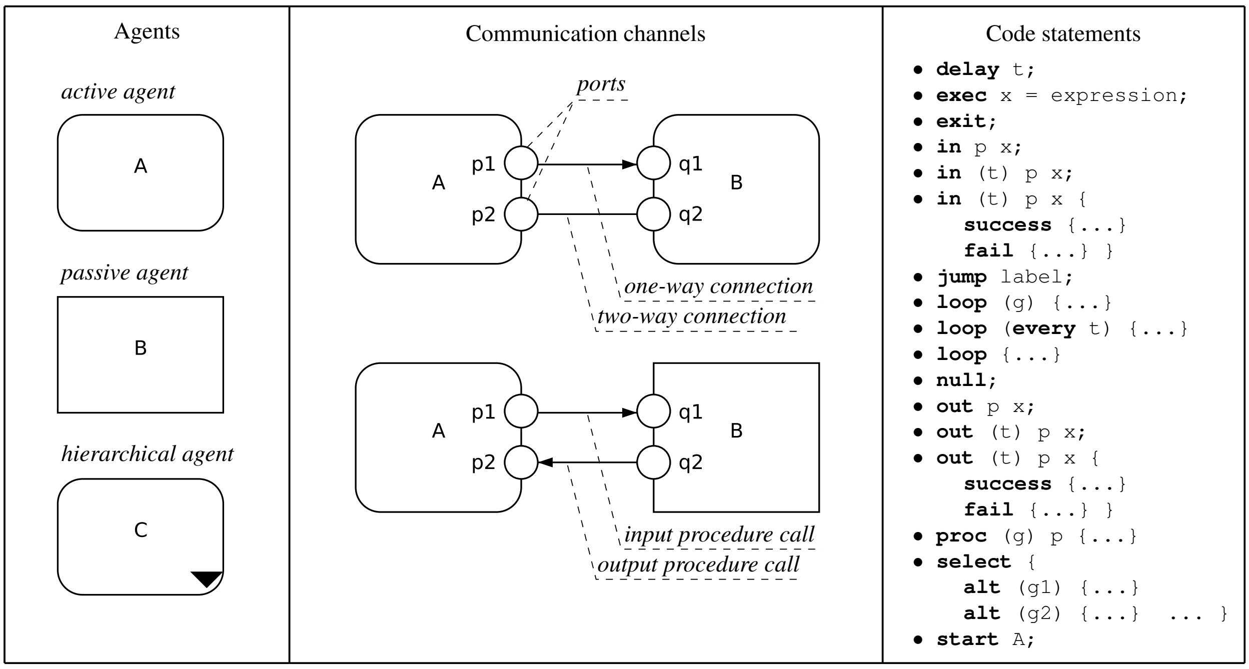
Introduction to Alvis Formal Modelling Language

Alvis [166, 167] is a formal modelling language intended for development and verification of systems consisting of concurrently operating units.

The language is being developed at AGH-UST in Kraków at the Department of Applied Computer Science. The main motivation behind the creation and development of Alvis language was to make the modelling and verification process more simple and accessible to software developers. In the proposed approach, the heavy mathematical foundations are hidden from the user without compromising the capabilities and expressive power of the formalism. Model description language is also very similar to the many popular programming languages, which further flattens the learning curve of its usage for developers. Alvis actually combines the advantages of formal methods and practical modelling languages. Main differences between Alvis and more classical formal methods, like Petri nets and process algebras, include the syntax that is more user-friendly from engineers' point of view, and the visual modelling language (communication diagrams) that is used to define communication among distinguished parts of a model called agents. The main difference between Alvis and industry programming languages is a possibility of formal verification of Alvis models using model checking techniques [10]. Alvis has its origins in the CCS process algebra [132, 1], the XCCS language [11, 164] and the Ada programming language [27]. The main result of this fact is the concept of agent that is borrowed from CCS. Agent denotes any distinguished part of the system under consideration with defined identity persisting in time. In contrast to process algebras, Alvis uses a high-level programming language to define behaviour of agents instead of algebraic equations. Moreover, the communication mechanisms used in Alvis are similar to the Ada rendezvous mechanism and calling entries of Ada protected objects [27].

In this chapter, the formal definition of the Alvis language is discussed. The mathematical aspects defined in the next sections of the chapter were the starting point in design of relevant algorithms, data structures, and development of the Alvis compiler tool, thoroughly discussed in the next chapter. The compiler design follows all definitions mentioned in this chapter. To make sure that compiler is following the design, exhaustive and automated testing of the code was aligned with the development. The ultimate goal of the compiler is to enable creation of the final *LTS* graph in compliance with the formal definitions.

The specification of the model of the system in the Alvis language consists of three levels of information details expressed in so-called layers: *graphical layer* (Sec. 3.1), *code layer* (Sec. 3.2), and *system layer* (Sec. 3.3). A diagram with an overview of the graphical and code layers is presented in Fig. 3.1. The system



```
• delay t;
• exec x = expression;
• exit;
• in p x;
• in (t) p x;
• in (t) p x {
    success {...}
    fail {...} }
• jump label;
• loop (g) {...}
• loop (every t) {...}
• loop {...}
• null;
• out p x;
• out (t) p x;
• out (t) p x {
    success {...}
    fail {...} }
• proc (g) p {...}
• select {
    alt (g1) {...}
    alt (g2) {...}  ... }
• start A;
```

Figure 3.1: Elements of Alvis modelling language

is modelled as a collection of agents that communicate with each other by exchanging arbitrary messages. The following details presented on the left and the middle part of the diagram are part of the *graphical layer*. There are three types of agents: active, passive, and hierarchical. *Active agent* perform some activities and can be treated as a thread of control in a concurrent or distributed system. *Passive agent* does not perform any individual activity, but provide a mechanism for the mutual exclusion and data synchronization and can be treated as shared memory or a disk drive. *Hierarchical agent* groups one or several agents into a logical subsystem and allows to modularise the design. An agent can communicate with other agents through ports. By default, an agent cannot communicate with any other agent. To make communication possible between two agents, an explicit communication channel between their ports must be defined in the graphical layer. Some ports of passive agents represent procedures (services) used to access shared data stored by the agents. Communication with a procedural port is treated as a procedure call.

The *code layer* defines the behaviour of every active and passive agent. The right part of the diagram makes an overview of elements with which the behaviour of an agent in Alvis can be described. Those are high-level instructions (code statements), variables (not presented), and procedures.

The last element of the Alvis model specification is *system layer*, which encapsulates the model of the underlying system that executes the code of the agents. It can specify, e.g. a number of processing unit available for active agents. However, the system layer is not expected to be designed by the user. It is just a configuration parameter set before the generation of the intermediate representation takes place.

The scheme of the modelling and verification process with Alvis is shown in Fig. 3.2. From a user's perspective, it starts from designing a model using prototype modelling environment called Alvis Editor. Then Alvis Compiler is used to translate it into Haskell source code, and its Haskell representation is used

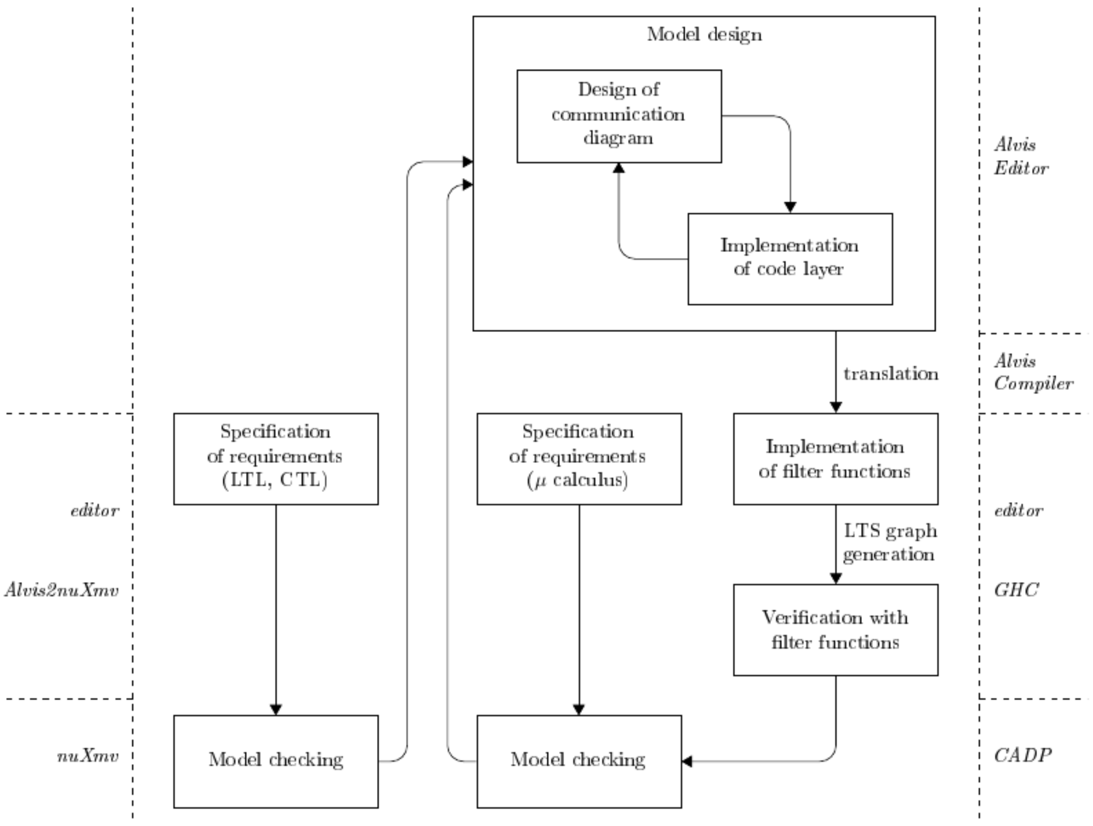

Model design
Design of communication diagram
Implementation of code layer
translation
Specification of requirements (LTL, CTL)
Specification of requirements (μ calculus)
Implementation of filter functions
LTS graph generation
Verification with filter functions
Model checking
Model checking
Alvis Editor
Alvis Compiler
editor
GHC
CADP
editor
Alvis2nuXmv
nuXmv

Figure 3.2: Modelling and verification process with Alvis

to generate the LTS graph (labelled transition system). The definition of the LTS graph is described starting from Sec. 3.4.

Alvis LTS graphs can be verified using the CADP toolbox [69]. It offers a wide set of functionalities, ranging from step-by-step simulation to massively parallel model-checking. Alvis Compiler provides a possibility to export an LTS graph into CADP Aldebaran format. We use the CADP evaluator tool to check whether an LTS graph satisfies requirements given as μ-calculus formulae [60, 96, 128]. It should be emphasized that this is an action-based approach. A μ-calculus formula focuses on actions labels, while states of the considered model are anonymized to straight index numbers.

A state-oriented approach is also provided by the nuXmv model checker [30] (the previous version known as NuSMV). The Alvis2nuXmv translator provides automatic translation of an Alvis LTS graph into an equivalent nuXmv state machine. The nuXmv tool enables automatic verification if formulae specified in a temporal logic (LTL, CTL or RTCTL) is satisfied by the model [10, 38].

Moreover, as Alvis compiler generates an intermediate representation of the LTS graph as a program and running that program creates the graph, it is possible to add arbitrary user-defined functions, e.g. to check invariants on-the-fly. Those type of functions are so-called *filtering functions*.

Lastly, the effort to add on-the-fly verification algorithm of LTS graph's properties expressed in weak LTL formulae was completed last year by Wojciech Kumoń [97]. The verification algorithm was implemented for a distributed version of LTS graph generation algorithm.

In this chapter theoretical aspects referring to modelling part of the process of verification of Alvis models are discussed in Sec. 3.1 – graphical layer of the model, Sec. 3.2 – code layer of the model, Sec. 3.3 – system layer of the model. Next, definitions related to the formal model are discussed in Sec. 3.4 – static part of the model, in Sec. 3.5 – dynamics of the model, and in Sec. 3.6 – LTS graph definition.

3.1 Graphical layer

The structure of communication connections between agents is designed using the graphical layer called *communication diagram* [165].

The *communication diagram* takes the form of a directed graph, with agents represented by the nodes and communication channels represented by the arcs. A bidirectional (duplex, two-way) communication channel between two active agents should be treated as a pair of arcs.

The concept of the communication diagram stems from the XCCS language diagram [11, 164]. The hierarchical dependencies in communication diagrams are an adapted concept of substitution transitions from coloured Petri nets [92, 162]. A part of a communication diagram can be placed at a separate page and be represented at the higher-level page by a single so-called hierarchical agent. In other words, a hierarchical agent represents a subsystem (module) of the considered system and the (sub)page attached to this agent describes the subsystem in details. Of course, such subpage may contain other hierarchical agents. Moreover, the same page may be attached to more than one hierarchical agent, so a designer may reuse some parts of the model. Therefore, the whole graphical layer of the model, i.e. the set of all pages, is a form of a directed acyclic graph of (sub)pages – in the more general case when at least one subpage is shared among two hierarchical agents. Alternatively, a form of a tree when no hierarchical agent has its subpage.

Hierarchical communication diagrams are a way to structure and ease maintenance of a design of a complex and distributed computation system. In Fig. 3.3, the substitution phase of a hierarchical agent with subpage of the designed system is presented. Thanks to the flexibility provided by the hierarchy, many

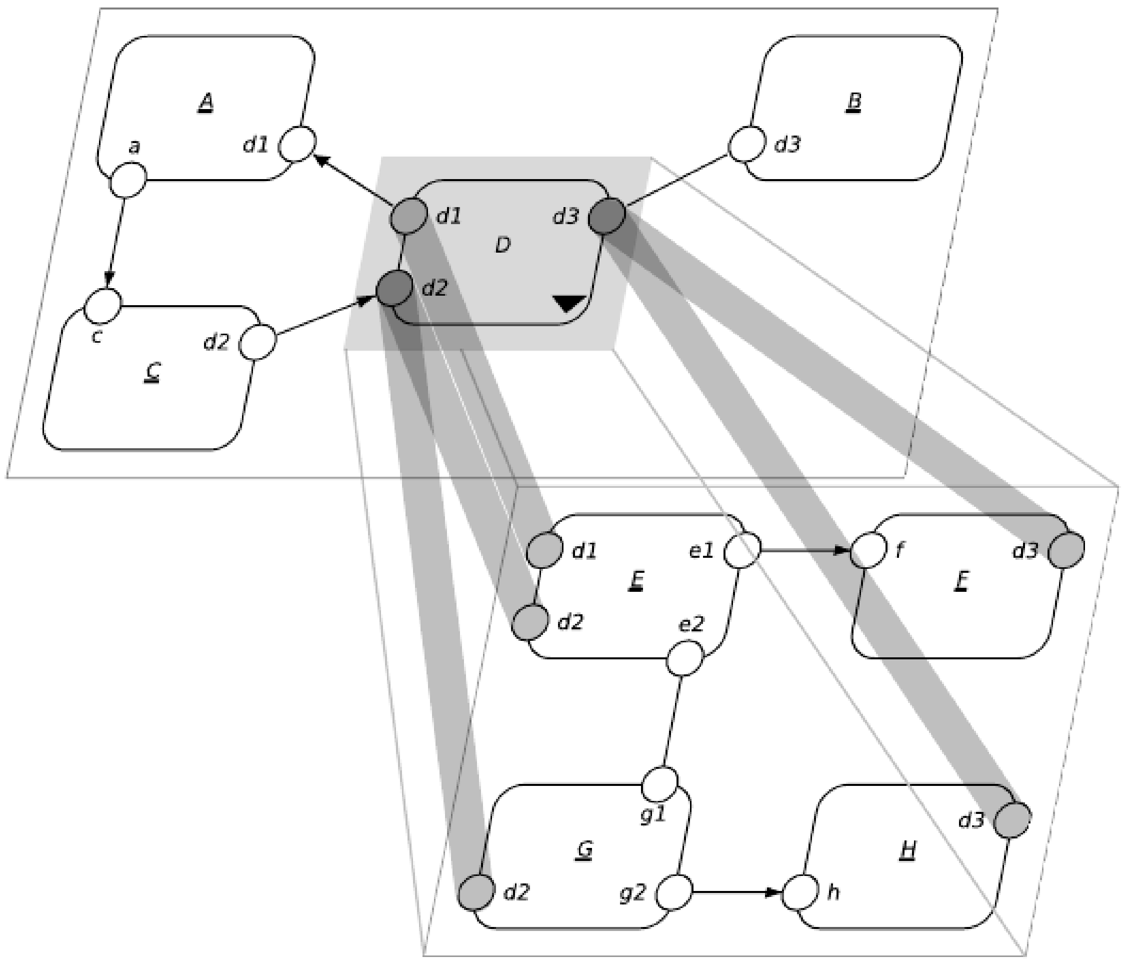


Figure 3.3: Extended hierarchical agent substitution by subpage

everyday tasks like updating the existing design by, e.g. increasing the number of computing units or addition to the design part of a different subsystem encapsulated in a separate module is an extremely easy task. The paper [165] provides the following patterns for the hierarchical diagrams:

- *Modules* – Splitting the system into smaller parts is one of the most basic concepts in software engineering. Fragments of functionality can be split into their respective modules, which encapsulate the separated pieces of logic. Then, working on a particular module does not require immediate consideration of the implications of the work on other parts of the system. Modularisation is invaluable for working efficiently. There are many other benefits of breaking the system into smaller modules, e.g. the model is more maintainable, testable and reusable. Moreover, such a module can be used as a reusable component. An example is presented in Fig. 3.4.
- *Removing Multiple Connections* – Creating even a moderately complex model can lead to situations in which the communication diagram becomes difficult to read due to the increasing amount of con-

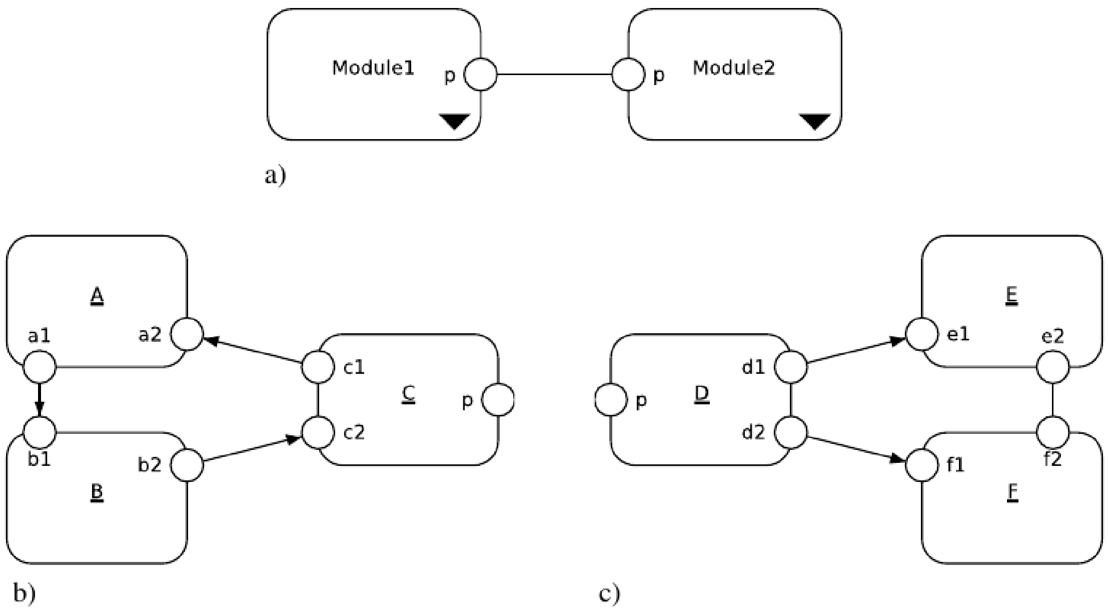


Figure 3.4: Grouping parts of the model into modules. a) Hierarchical agents for two modules; b) subpage for agent Module1; c) sub- page for agent Module2

nections between the agents. The warning signal is the moment in which some of the connections are intersecting each other. If a port(s) of an agent is connected with multiple other agents ports, the connection can be simplified by the introduction of a hierarchical agent. In Fig. 3.5, ports p and q were connected with two other ports.

- *Replacing Multiple Agent's Instances with a Single Representation* – When the model contains multiple instances of the same agent, the repeated agent can be replaced with a hierarchical agent. Repeated elements are a prevalent scenario in real-world modelling systems. Elements such as sensors, indicators, displays and other sub devices are often repeated in the scope of a single real-time system. In the case of computational systems, the repeated elements could be threads, processes, instances of the machines, databases, and others. Each of them can be on some level of abstraction, represented by a separate hierarchical agent. New instances can be easily added by copying an instance on the subpage. The advantage is that all the connections will be automatically added by the algorithm that flattens the hierarchy. In Fig. 3.6, agent B with 3 instances is represented as a single hierarchical agent.
- *Grouping Repeating Fragments of a Model* – In extensive systems, repeating fragments of the model can often occur. Each of these excerpts represents similar functionality. The only difference is that they are placed in a different part of the system and are connected to different agents. Such a model is therefore potentially easy to disrupt. Assuming that these fragments need some adjustment or modification, one would have to find all of these fragments and update them one by one. The solution to this problem is once again in a hierarchy. In Fig. 3.7, agents D and E represent the subsystem that has to

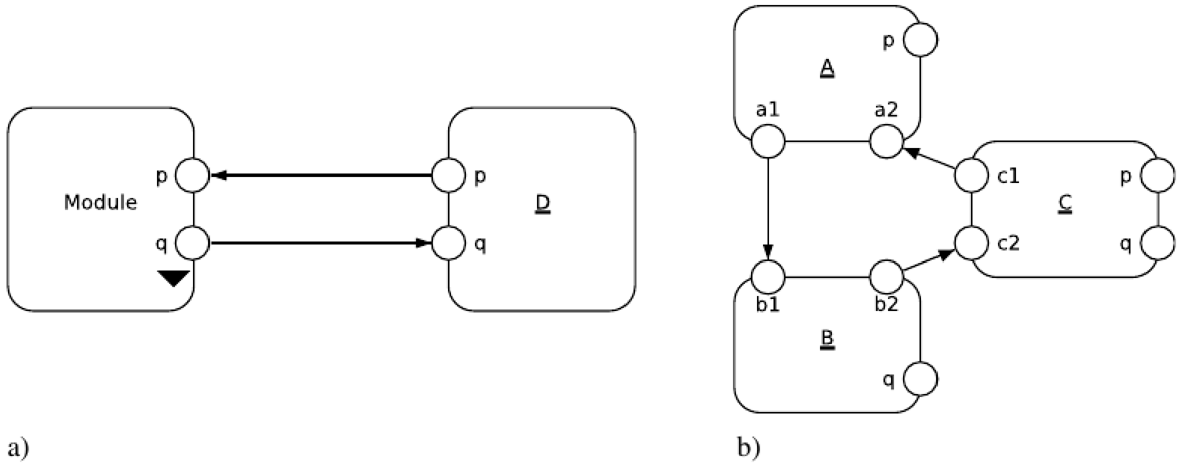


Figure 3.5: Removed multiple communication channels. a) Inserted hierarchical agent Module; b) subpage for agent Module

plugged into two places (possibly with slight modifications in the code layer). The instances of agents D and E are then grouped on a single page of hierarchical agent DE.

Hierarchical agents are beneficial from the design perspective. However, from the Alvis compiler perspective, only non-hierarchical communication diagram is required. The paper [165] also discusses the algorithm of flattening the hierarchy into a single page – non-hierarchical communication diagram. Before the compilation of the model will happen the preprocessing phase is run which flattens the diagram into the required non-hierarchical form. Therefore, only the non-hierarchical communication diagram will be formalised below, and the interested reader is advised to refer to the paper [165] for discussion of hierarchy in more detail.

Let us start with the following definitions: $\mathcal{P}(X)$ denotes the set of ports of an agent X. We can distinguish some subsets of $\mathcal{P}(X)$:

$\mathcal{P}_{in}(X)$ ($\mathcal{P}_{out}(X)$) denotes the set of input (output) ports of X. An input (output) port is a port with at least one one-way connection leading to (from) the port or with at least one two-way connection.

- $\mathcal{P}_{unc}(X) = \mathcal{P}(X) \setminus (\mathcal{P}_{in}(X) \cup \mathcal{P}_{out}(X))$ denotes the set of unconnected ports.
- $\mathcal{P}_{proc}(X)$ denotes the set of procedural ports of passive agent X i.e. ports with defined the **`proc`** statement.

For a set of agents W, we define: $\mathcal{P}(W) = \bigcup_{X \in W} \mathcal{P}(X)$, $\mathcal{P}_{in}(W) = \bigcup_{X \in W} \mathcal{P}_{in}(X)$, etc. Moreover, let $\mathcal{P}$ denote the set of all model ports, $\mathcal{P}_{in}$ denote the set of all model input ports, etc. We use two notations for ports. A single lower-case letter, e.g. p denotes a port p of an agent. If it is necessary to point out the agent, the dot notation is used, e.g. $X.p$.

Definition 3.1.1. *Non-hierarchical communication diagram* is the triple $D = (\mathcal{A}, \mathcal{C}, \sigma)$, where:

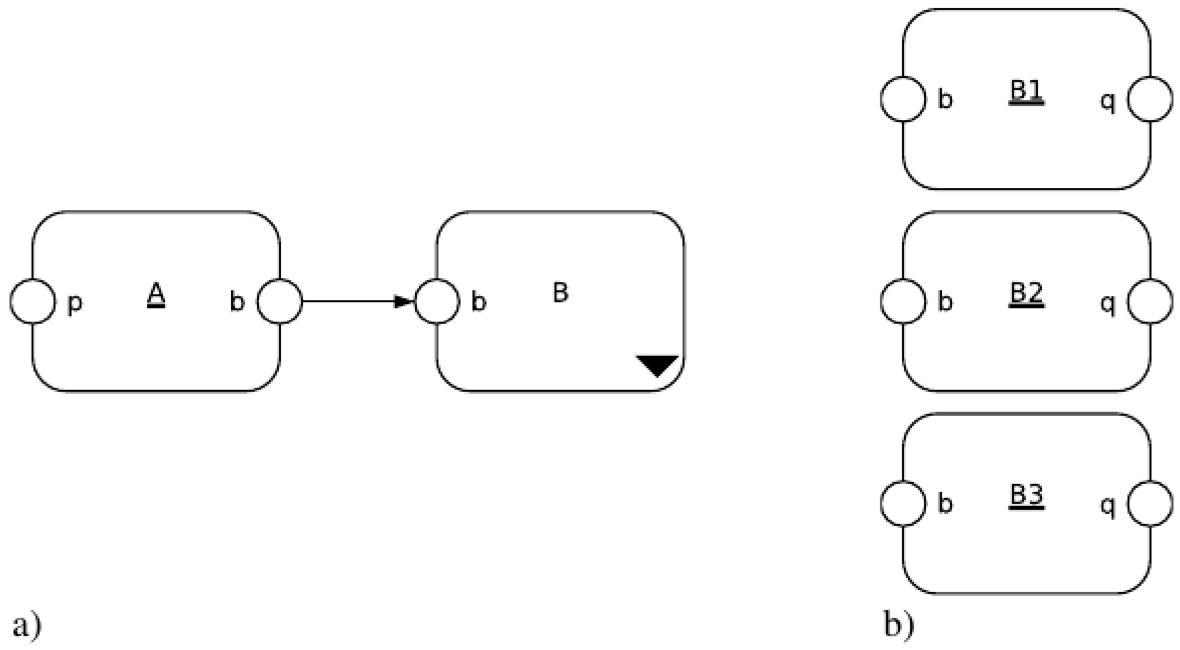


Figure 3.6: Simplifying addition/removal of new instances of particular agent. a) Inserted hierarchical agent B; b) subpage for agent B

- $\mathcal{A} = \{X_1, \ldots, X_n\} = \mathcal{A}_A \cup \mathcal{A}_P$ and $\mathcal{A}_A \cap \mathcal{A}_P = \varnothing$ is a set of agents consisting of two disjoint sets of active agents ($\mathcal{A}_A$) and passive agents ($\mathcal{A}_P$).
- $\mathcal{C} \subseteq \mathcal{P} \times \mathcal{P}$ is the communication relation that is restricted by the following conditions:
  - One cannot define links between ports that are owned by the same agent.

$$\forall_{X \in \mathcal{A}} \colon (\mathcal{P}(X) \times \mathcal{P}(X)) \cap \mathcal{C} = \varnothing \tag{3.1}$$

  - Procedural port is either an input port or output port. Therefore, every procedure has fixed flow of the sending parameter to (**input procedure**) or from (**output procedure**) it.

$$\mathcal{P}_{proc} \cap \mathcal{P}_{in} \cap \mathcal{P}_{out} = \varnothing \tag{3.2}$$

  - Connection between an active agent and a passive agent is required to use one of the passive agent's procedural ports and corresponds to a call to that procedure (service) and the procedure is executed in the context of the active agent.

$$(p, q) \in (\mathcal{P}(\mathcal{A}_A) \times \mathcal{P}(\mathcal{A}_P)) \cap \mathcal{C} \implies q \in \mathcal{P}_{proc} \tag{3.3}$$

$$(p, q) \in (\mathcal{P}(\mathcal{A}_P) \times \mathcal{P}(\mathcal{A}_A)) \cap \mathcal{C} \implies p \in \mathcal{P}_{proc} \tag{3.4}$$

  - Connection between two passive agents cannot be realized as connection between two procedu-

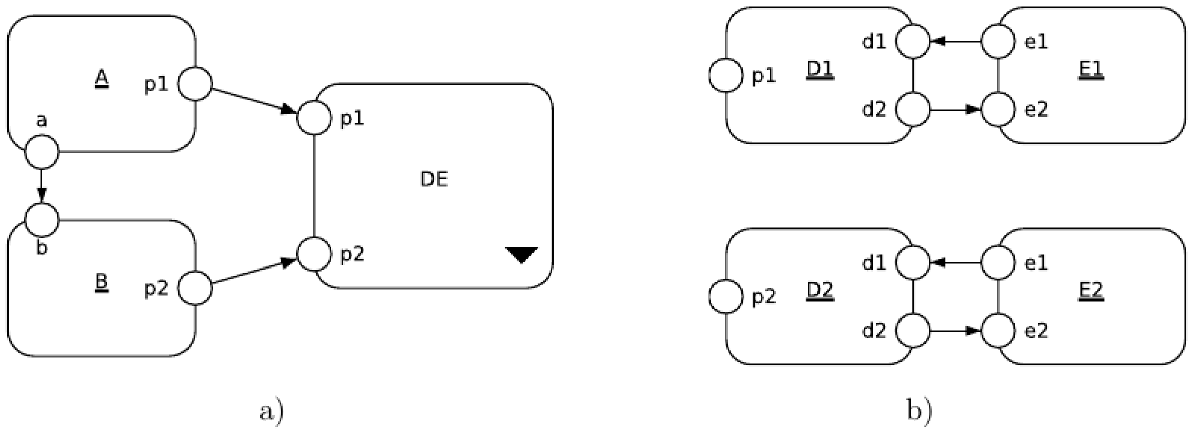


Figure 3.7: Removed multiple communication channels. a) Inserted hierarchical agents DE; b) subpage for agent DE

ral ports.

$$(p, q) \in (\mathcal{P}(\mathcal{A}_P) \times \mathcal{P}(\mathcal{A}_P)) \cap \mathcal{C} \wedge p \neq q \implies \implies (p \in \mathcal{P}_{proc} \wedge q \notin \mathcal{P}_{proc}) \vee (q \in \mathcal{P}_{proc} \wedge p \notin \mathcal{P}_{proc}) \tag{3.5}$$

- $\sigma : \mathcal{A}_A \longrightarrow \{Flase, True\}$ is **start function** that indicates which active agents are activated in the initial state (**True** – active agent is in e**X**ecuting mode, **False** – active agent is in **I**nitial mode and can be later started by another agent).

As an example of the application of the definition 3.1.1, one can consider the Producer–Consumer problem 1.9:

$$\begin{aligned}
\mathcal{A}_A &= \{Producer,\ Consumer\},\\
\mathcal{A}_P &= \{Buffer\},\\
\mathcal{P}_{in} &= \{Consumer.pull, Buffer.put\},\\
\mathcal{P}_{out} &= \{Producer.push, Buffer.get\},\\
\mathcal{P}_{proc} &= \{Buffer.put, Buffer.get\},\\
\mathcal{C} &= \{(Producer.push, Buffer.put), (Buffer.get, Consumer.pull)\},\\
\sigma(Producer) &= True,\\
\sigma(Consumer) &= True.
\end{aligned}$$

3.2 Code layer

The definition of every individual agent behaviour is described in the model's *code layer* [172, 163].

The *code layer* takes the form of the program source code that is a mixture of Alvis code statements and Haskell expressions and types [142, 143, 52, 116]. The grammar of the Alvis language is too broad to discuss it and can be found in the Alvis compiler code repository. Therefore, we focus here on the essential elements of the code layer and its descriptions are presented here in a less formal way: as an example and their short description. That concise way will be easier to follow, and the interested reader is advised to look in the Appendix for further details and formal definitions.

From the high-level elements of the code layer down to more detailed one, Alvis code layer is a series of individual Agents definitions. As it was already mentioned, there are three types of agents in Alvis modelling language: active agent, passive agent, and hierarchical agent. The last one type – a hierarchical agent is a grouping element from *graphical layer* perspective, and it is not reflected in the *code layer*.

The definition of an active agent consists of the agent name, list of agent's local variables and a main code block of instructions. Listing 3.1 presents the definition of $Producer$ – an active agent with single variable $value$ and a 3 instruction long main code block.

```
agent Producer {
 -- local variables
 value :: Int = 0;
 -- main code block
 loop {
  value = pick [1..3];
  out push value;
 }
}
```

Listing 3.1: Active agent code definition from Producer–Consumer problem

The definition of a passive agent consists of agent's name, list of agent's local variables and list of procedures. Listing 3.2 presents the definition of $Buffer$ – a passive agent with two variables $value$, and $isFull$ and two procedures get, and put.

```
agent Buffer {
 -- local variables
 value :: Int = 0;
 isFull :: Bool = False;

 -- procedures
 proc (isFull) get {
  out get value;
  isFull = not isFull;
  exit;
 }

 proc (not isFull) put {
  in put value;
```

```
    isFull = not isFull;
    exit;
  }
}
```

Listing 3.2: Passive agent code definition from Producer–Consumer problem

So the main difference between the active agent and passive agent definitions is that active agent has only one code block of instructions while a passive agent can have more than one entry point of execution that are its procedures.

It is possible that two or more agents have the same behaviour definitions. Then instead of duplicating code definitions of the agents in the Alvis code layer, one can replace the name of the agent, in its code definition, with a comma-separated list of names of all agents sharing the same behaviour. In Listing 3.3, the example of three agents $FirstConsumer$, $SecondConsumer$, and $ThirdConsumer$ is presented.

```
agent FirstConumser, SecondConsumer, ThirdConsumer {
  -- local variables
  value :: Int = 0;

  -- main code block
  loop {
    in pull value;
  }
}
```

Listing 3.3: Three agents with the same definition (adapted Producer–Consumer problem)

Lastly, the list of all instructions that are available in the Alvis model with system layer α^0 is presented below:

- **`delay`** `t;`

  Delays execution of the context agent by t time units.

- **`exec`** `x = expression;`

  `x = expression;`

  Evaluates the given $expression$ and assigns the result to the x variable.

- **`exit;`**

  Terminates execution of the active agent or passive agent's procedure.

- **`in`** `p;`

  Blocks the execution of the context agent up to receiving the signal from the p port (by receiving signal from procedure of passive agent, we understood, here and in the below descriptions, the completion of the execution of the passive agents procedure).

- **`in`** `p x;`

Blocks the execution of the context agent up to receiving the value from the p port and stores it to the x variable

- **in** (t) p;

  Blocks the execution of the context agent up to receiving the signal from the p port, but for no more than t time units. If the time runs out, the agent continues its execution with the next instruction after the **in** statement.

- **in** (t) p x;

  Blocks the execution of the context agent up to receiving the value from the p port, but for no more than t time units. If the time runs out, the agent continues its execution with the next instruction after the **in** statement.

-
  ```
  in (time) p {
    success {
      -- instructions
    }
    fail {
      -- instructions
    }
  }
  ```

  Blocks the execution of the context agent up to receiving the signal from the p port, but for no more than t time units. In the case of successful communication, the execution is resumed from the first instruction of the **success** block. In the case of timed out communication, the execution is resumed from the first instruction of the **fail** block. The order of the **success** and **fail** blocks can be arbitrary, and one of them can be omitted. If the code block is not present, the first instruction to be executed after execution is resumed is the first instruction after the statement **in**. If both blocks **success** and **fail** are to be omitted, one can use the syntax from the previous line.

-
  ```
  in (time) p x {
    success {
      -- instructions
    }
    fail {
      -- instructions
    }
  }
  ```

Blocks the execution of the context agent up to receiving the value from the p port and stores it to the x variable but for no more than t time units. In the case of successful communication, the execution is resumed from the first instruction of the block **`success`**. In the case of timeout communication, the execution is resumed from the first instruction of the block **`fail`**. The order of the blocks **`success`** and **`fail`** can be arbitrary, and one of them can be omitted. Then, the first instruction to be executed after execution is resumed is the first instruction after the statement **`in`**.

- **`jump`** `label`

  The execution of the code progresses unconditionally from the instruction with the $label$ label.

-
  ```
  loop (g) {
    -- instructions
    null;
  }
  ```

  The g guard condition is evaluated. If the result is true, then the first instruction inside **`loop`** block is executed. Otherwise, the first instruction after the whole loop statement is executed. When the last instruction of the **`loop`**'s block is executed, then the execution is transferred back to the instruction **`loop`**. It is required that the last instruction in the **`loop`**'s block is a **`null`** instruction.

-
  ```
  loop {
    -- instructions
    null;
  }
  ```

  This is the unconditional variant of the **`loop`** statement. Therefore, the next instruction to be executed is the first instruction in the block **`loop`**. When the last instruction of the block **`loop`** is executed, then the execution is transferred back to the **`loop`** instruction. It is required that the last instruction in the **`loop`**'s block is a **`null`** instruction.

-
  ```
  loop (every t) {
    -- instructions
    null;
  }
  ```

  This is the variant of the **`loop`** statement with timer. Therefore, the next instruction to be executed is the first instruction in the block **`loop every`**. When the last instruction of the **`loop every`**'s block is executed, then the execution is suspended for the left t time units before the execution is transferred back to the instruction **`loop every`**. It is required that the last instruction in the **`loop`**

↪ **every** block is a **null** instruction.

- **null**;

  Empty instructions, do nothing.

- **out** p;

  Blocks the execution of the context agent up to completion of sending the signal by the p port (by sending signal to procedure of passive agent, completion is understood as finish of the execution of passive agents procedure).

- **out** p x;

  Blocks the execution of the context agent up to completion of sending the value by the p port read from the variable x.

- **out** (t) p;

  Blocks the execution of the context agent up to completion of sending the signal by the p port, but for no more than t time units. If the time runs out, the agent continues its execution with the next instruction after the **out** statement.

- **out** (t) p x;

  Blocks the execution of the context agent up to completion of sending the value by the p port read from the x variable, but for no more than t time units.

-
  ```
  out (time) p {
    success {
      -- instructions
    }
    fail {
      -- instructions
    }
  }
  ```

  Blocks the execution of the context agent up to completion of exchanging the signal through the p port, but for no more than t time units. In the case of successful communication, the execution is resumed from the first instruction of the **success** block. In the case of timeout communication, the execution is resumed from the first instruction of the **fail** block. The order of the **success** and **fail** blocks are arbitrary and one of them can be omitted. If the code block is not present, the first instruction to be executed after execution is resumed is the first instruction after the **out** statement. If both blocks **success** and **fail** are to be omitted, one can use the syntax from the previous line.

-
  ```
  out (time) p x {
  ```

```
  success {
    -- instructions
  }
  fail {
    -- instructions
  }
}
```

Blocks the execution of the context agent up to receiving the value from the p port and stores it to the x variable, but for no more than t time units. In the case of successful communication, the execution is resumed from the first instruction of the **success** block. In the case of timeout communication, the execution is resumed from the first instruction of the **fail** block. The order of the **success** and **fail** blocks are arbitrary and one of them can be omitted. Then, the first instruction to be executed after execution is resumed is the first instruction after the **out** statement.

•
```
select {
  alt (g1) {
    -- instructions
  }
  alt (g2) {
    -- instructions
  }
  -- more guarded blocks
}
```

In Alvis for the simplicity reasons, there is only a single choice statement namely **select**. The statement contains from one up to an arbitrary number of guarded blocks of code that are called *branches*. A branch that guard is evaluated to **True** is called *open*, otherwise it is called *closed* branch. The branches' guards are evaluated sequentially. Only the first open branch is executed. When there is no open branch the first statement after **select** is executed.

• **start** A;

Activates the agent A. If the state of the active agent A is **I**nit then its state is changed to e**X**ecuting. Otherwise, the instruction has no effect. The argument of the **start** instruction has to be the name of the active agent within the model.

3.3 System layer

The third and the last element of the Alvis model specification is *the system layer*. The system layer on the contrary to the two previous layers, namely graphical and code ones, is not intended to be highly configurable and detailed by the designer of the model. The role of the system layer is to abstract away the different computation execution models, validate assumptions regarding the type of failures that can happen and encapsulates the differences in the computation resources available under the simple configuration parameter as the chosen system layer. Choosing the particular system layer settings affects the LTS generation, as different states of the system are reachable.

In the literature, one can encounter the three most common system models describing distributed computing systems by the timing issues that can happen in that kind of systems [94]:

- Synchronous model – In such system model, there is a known fixed upper bound Δ on time required for a message to be sent from one computing unit to another and a known fixed upper bound Φ on the relative processes pauses and drifts of internal clocks.
- Asynchronous model – In such system model, there are no bound Δ nor Φ that would limit a priori exchanging time of the messages nor relative drifts of internal clocks of processing units. However, in such a model, it was proven that in the presence of faults of a computing unit or network/message buses there is not possible to achieve consensus [62]. The model is unrealistically generic [94].
- Partially synchronous model – This system model is something between synchronous and asynchronous model, and it is not a single model [57]. Some authors define it like that Δ and Φ exist but are unknown, in another definition of system model Δ and Φ are known but are only guaranteed to hold after some unknown time T. This model is closer to reality as, e.g. Internet network connecting servers (computing units) is an unreliable carrier, and there are no upper-bound guarantees that packets will be delivered within.

However, Alvis is not limited to only distributed computing systems, and various other models can be successfully designed within Alvis toolset. Therefore, at the moment of writing of the dissertation, there are available two system layers for the user of Alvis:

- α^0 – where every agent has access to its computing unit, and therefore scheduling effects can be neglected.
- α^1 – where there is only one computing unit available, and agents compete between each other to access it. That layer was discussed in the PhD dissertation of J. Baniewicz [12].

Additionally, in the α^0 layer there are again two versions of interpretation of the code, namely: **timed** and **non-timed** version. The timed version of Alvis corresponds to the synchronous model, while the non-timed version of Alvis corresponds to an asynchronous model. In the timed version, one can specify how long it takes to perform every Alvis command explicitly for every computing unit. In the non-timed version, there is no notion of time at all. This dissertation focuses on the α^0 layer and non-timed version of Alvis.

However, non-timed version of Alvis is of type of asynchronous model, there are possibilities to limit generated paths that would be unrealistic by leveraging fairness assumptions with temporal logics (this is however outside of the scope of the dissertation).

Moreover, additionally to the timing issues, one can consider nodes failure (processor unit failures that execute active agent code) [94]:

- Crash-stop faults – when failure of its processing unit happens, an agent would stop and will never come back to execute code again.
- Crash-recovery faults – when failure of its processing unit happens, an agent would stop and can start over after some time. It is assumed that local variables could be lost (forgotten), but there is access to non-volatile memory storage.
- Byzantine (arbitrary) faults – there is no assumption on the type of faults that can happen in the system. It can even happen that an agent tries to trick others. The problem was described as Byzantine general problem [103].

None of the above failure types is at this moment formalised and configurable out of the box in Alvis, e.g. by using a flag of the compiler. However, it is straightforward to add *enabling* and *firing* rules with generic conditions that will allow introducing both crash-stop and crash-recovery faults into the Alvis. It is even easier to modify the Intermediate Haskell Representation of the model (generated by the compiler to introduce those type of failures) without the need to modify Alvis toolset at all!

3.4 Alvis formal model

In the former sections, the layers of Alvis model that allows designing specific system were introduced and described in details. Those layers allow to design the distributed system, but are not suitable to perform model checking and verification tasks. In order to do that, formal definition of Alvis model has to be introduced and the rules of translation of code layer 3.2 and graphical layer 3.1 into the formal model has to be provided. The rules depend on the system layer version chosen by the designer 3.3. As it was already mentioned in the former section, the α^0 system layer is only considered, as α^1 is covered in the Baniewicz's dissertation [12, 13].

As Alvis is an explicit state model verification language, the modelled system state space is formally described as a directed graph. The vertices of the graph contain a particular state of the system, while edges correspond to the execution of Alvis code commands or events that appear in the system. This section describes the static part of the formal model. By "static", it is understood how the states of the Alvis model are defined, i.e. the vertices of the graph. Next section 3.5 will deal with a dynamic part of the model and therefore, the edges will be described and the rules that govern graph creation.

To begin with, we start from the top-level description, and then we will go into details. Therefore, the definition 3.4.1 formalizes the definition of Alvis model as specified by its graphical, code and system layer.

Definition 3.4.1. The *model* in Alvis language is the triple $\mathbf{A} = (D, B, \varphi)$, where:

- $D = (\mathcal{A}, \mathcal{C}, \sigma)$ is a *non-hierarchical communication diagram* cf. Def. 3.1.1.
- B is a syntactically correct *code layer* cf. sec. 3.2.
- φ is a *system layer*.

Next, we can specify what we mean by formal model state, i.e. a single element from the model's state space. The definition 3.4.2 formalizes that a system state is a tuple of states of particular agents. Graphically system state space is presented in Fig. 3.8.

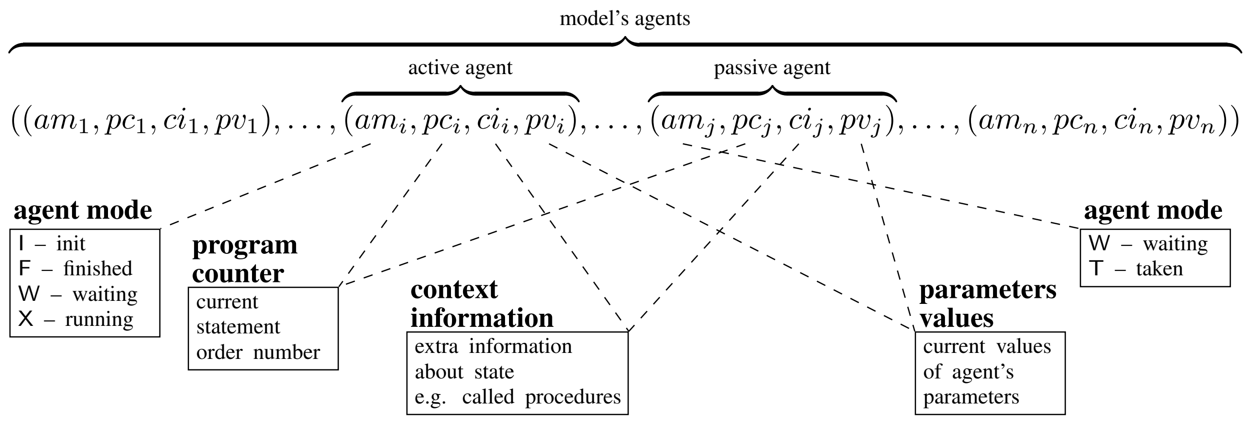

model's agents
active agent
passive agent
$((am_1, pc_1, ci_1, pv_1), \ldots, (am_i, pc_i, ci_i, pv_i), \ldots, (am_j, pc_j, ci_j, pv_j), \ldots, (am_n, pc_n, ci_n, pv_n))$
agent mode
I – init
F – finished
W – waiting
X – running
program counter
current statement order number
context information
extra information about state e.g. called procedures
parameters values
current values of agent's parameters
agent mode
W – waiting
T – taken

Figure 3.8: Representation of an Alvis model state

Definition 3.4.2. A *state* of a model $\mathbf{A} = (D, B, \alpha^0)$, where $D = (\mathcal{A}, \mathcal{C}, \sigma)$ and $\mathcal{A} = \{X_1, \ldots, X_n\}$ is a tuple,

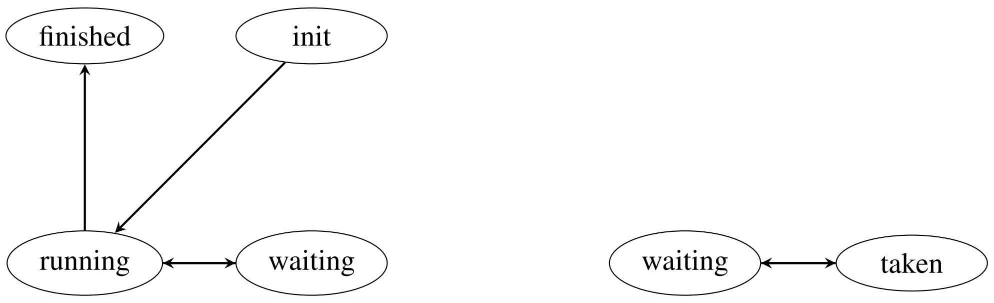
$$S = (S(X_1), \ldots, S(X_n)), \tag{3.6}$$

where $S(X)$ is the state of agent X.

By state of particular agent, cf. Definition 3.4.3, we mean the values of agent mode, program counter, its whole context list, and values of all its local variables. This approach is typical to the explicit state model checkers, where state is kept explicitly.

Definition 3.4.3. The *Agent X's state* is the 4-tuple $S(X) = (am(X), pc(X), ci(X), pv(X))$, where:

- $am(X)$ is agent X's mode.
- $pc(X)$ is agent X's program counter value.
- $ci(X)$ is agent X's context information list.
- $pv(X)$ is agent X's parameters (variables) tuple.



(a) active agent's mode transitions **(b)** passive agent's mode transitions

Figure 3.9: All the possible transitions between modes of the agent in respect to the type of the agent

We have to define what we mean by all those elements. Let us start by deep diving into agent mode. Agent mode describes the current state of the control thread related to the active agent or state of the passive agent lock related to mutual exclusion mechanism of accessing its procedures. All possible values of *the agent's mode*, i.e. the values of the function $am(X)$ are listed below. The presented modes are available in the system layer α^0. Some modes are allowed only for passive or only for active agents. Moreover, the modes cannot be changed arbitrarily, and the possible transitions between agent modes are restricted to the ones presented in Fig. 3.9.

- e**X**ecuting (running or X) – It is available only for active agents. It signifies that the agent is executing its instructions or instructions of a called procedure of the passive agent. The agent can be activated from the start in that mode when the activation function states so cf. Def. 3.1.1.
- **F**inished – (F) It is available only for active agents. It signifies that the agent finished its execution by executing its last instruction with no following instruction or by executing **`exit`** instruction explicitly. When an agent is in the **F**inished mode, there is no instruction to change its mode again.
- **I**nit (initial or I) – It is available only for active agents. This is the default mode of the active agent in Alvis model. The active agent A can transit into the running mode when any other already running agent executes **`start`** `A` instruction.
- **T**aken (T) – It is available only for passive agents. It signifies that one of the procedures of the passive agent is executed in the context of an active agent (possibly indirectly). So it signifies that lock was acquired and no other agent can execute the code of that passive agent.
- **W**aiting (W) – It is the only mode available for both active and passive agents. In the context of an active agent, it signifies that its execution was suspended, e.g. agent waits for an inaccessible procedure or time-out event. In the context of a passive agent, it means that the passive agent is available and no other agent is executing its procedures. That means the lock-related to mutually exclusion is free to be acquired. When a passive agent is in the W mode its context list contains all the available procedures to be called. Procedure may have conditional expression, so-called guard

expression, that enables the procedure to be called or disables it. The passive agent has to be in the W mode in the initial state of the system, as none of the locks could already be acquired.

Next element of the tuple describing agent state is function $pc(X)$ – *The agent's program counter* value. Program counter indicates the instruction to be executed by the agent X or the instruction that execution was recently started but not finished due to unavailability of some resource. The former case is reflected by the change of context (active) agent mode change to W mode. The value of the program counter is given the following meaning:

- $pc(X)$ indicates the instruction **`delay`** iff the next instruction to be executed is the instruction **`delay`** or **`delay`** was already executed, and the active agent or the context agent is now waiting for the time out event to continue the execution.
- $pc(X)$ indicates the instruction **`exec`** (**`exit`**, **`jump`**, **`null`**, **`start`**) iff the next instruction to be executed is the instruction **`exec`** (**`exit`**, **`jump`**, **`null`**, **`start`**).
- $pc(X)$ indicates the instruction **`loop`** iff the next action to be performed is the evaluation of the condition of the instruction **`loop`** and advancing pc based on the result of the condition evaluation. If the condition is satisfied, then the $pc(X)$ is advanced to the first instruction inside the **`loop`**. Otherwise, the program counter is set to the first instruction related to the next statement after the **`loop`**. When there is an unconditional **`loop`** then it is likewise the **`loop`** with the condition that is always true.
- $pc(X)$ indicates the instruction **`loop every`** iff the next action to be performed is starting the execution of the **`loop every`**. The $pc(X)$ is advanced to the first instruction inside the **`loop`** ↪ **`every`**'s block of code.
- $pc(X)$ indicates the instruction **`select`** iff the next action to be executed is the execution of all guards of the branches of the instruction **`select`**. If at least one condition is satisfied, then the $pc(X)$ is advanced to the first instruction in the first open branch of the **`select`** statement. Otherwise, the program counter is set to the first instruction related to the next statement after the **`select`**.
- $pc(X)$ indicates the instruction **`in`** (**`out`**) iff the next instruction to be executed is the instruction **`in`** (**`out`**) or **`in`** (**`out`**) was already executed, and the active agent or the context agent is waiting for the communication finalization, end of the procedural call or the time-out event to resume the execution.

There is another restriction posed on the value of agent mode and value of its program counter. There is a functional dependency on mode and value of the program counter, which is presented in table 3.1. There is one group where an agent cannot execute code, and therefore the value of its program counter has to be equal 0 (a special value that does not indicate any instruction) and another group of modes where program counter has to indicate existing command (non-zero – a positive value of program counter).

Table 3.1: Dependency between agent's mode and agent's program counter value

$am(X)$	$pc(X)$
F	0
I	0
X	current instruction
T	current instruction
W (active agent)	current instruction
W (passive agent)	0

The third value of agent's state is a function $ci(X)$ – the agent's *context list* that contains additional information about the events that the agent X awaits, the procedure being executed by the agent or available procedures to be called. The following entries are possible:

- $proc(p)$ – This value represents the called procedure. The argument corresponds to the called procedural port's name.
- $in(p)$ – When a passive agent is in the *wainting* mode this entry informs of the available for call input procedural port of that agent. When a passive agent is in the *taken* mode or an active agent is in the *waiting* mode it indicates a suspended, not yet completed communication and that agent waits for the peer agent to be available again. In this case, the argument p tells the name of the port of the agent on which communication is awaited.
- $out(p)$ – When a passive agent is in the *wainting* mode, this entry informs of the available for call output procedural port of that agent. When a passive agent is in the *taken* mode or an active agent is in the *waiting* mode, it indicates a suspended communication and that agent waits for the peer to be available. In this case, the argument p tells the name of the port of the agent on which communication is awaited.
- $timer(n, t)$ – This value represents the timer counting down the time left to the time-out event of the currently executed statement limited to **`loop every`**, **`delay`**, and non-blocking **`in`**/**`out`**. The first argument n is the index of the statement to which the timer refers to and the second argument t represents the number of time units left until time-out event occurs for that statement and it is always positive.
- $timeout(n)$ – This value represents a time-out signal generated by the timer reaching 0 time-units. At the moment when the timer reaches 0, its entry is replaced with the $timeout$ entry. The value signifies for the instruction of **`loop every`** that the loop should be resumed as soon as possible. For the instruction **`delay`**, it signifies that the execution of the agent should be resumed with the next instruction. For the non-blocking statement **`in`**/**`out`**, it signifies that the instruction should be aborted

and the execution resumed from the failed block of code.

- $sft(t)$ – This value is an acronym of the context info: *step finish time* and represents the number of time units left to complete the current instruction being executed.
- $lock$ – This value represents that the agent is locked and is available only to the agent with whom it started a communication.

The last value of agent's state is $pv(X)$ function – the agent's local *variable* values that are a tuple of all variables and holds information of all their values in a given state. Values of the function are agent-specific and therefore can vary between different agents. If an agent has no local variables, 0-tuple denotes that fact.

Once we have discussed the structure of the state of the system, we will discuss shortly the valuation of one particular state of the system, namely initial state. The initial state is the state form which the evolution of the system begins. The initial state for a model is specified in the code layer. A designer is required to provide initial values of the variables and valuation of the activation function. Furthermore, a few general rules govern context lists population and program counter initialisation. Formal definition of the initial state is understood as Definition 3.4.4.

Definition 3.4.4. Let an Alvis model $\mathbf{A} = (D, B, \alpha^0)$, where $D = (\mathcal{A}, \mathcal{C}, \sigma)$ and $\mathcal{A} = \{X_1, \ldots, X_n\}$ be given. The *initial state* S_0 is defined as follows:

- $am(X)$ = X (*running*), for any active agent X such that $\sigma(X) = True$;
- $am(X)$ = I (*init* – idle), for any active agent X such that $\sigma(X) = False$;
- $am(X)$ = W (*waiting*), for any passive agent X;
- $pc(X) = 1$ for any active agent X in the *running* mode and $pc(X) = 0$ for other agents.
- $ci(X) = [\,]$ for any active agent X;
- $ci(X)$ contains names of accessible procedures for any passive agent X;
- For any agent X, $pv(X)$ contains X parameters with their initial values as designed in the code layer 3.2.

As an example, we can consider initial state of the Producer–Consumer model. The initial state of the system is the 3-tuple of initial states of all agents.

$$S_0 = (S_0(Producer), S_0(Buffer), S_0(Consumer))$$

The initial state of the agent $Producer$ is a 4-tuple, where $am(Producer) = X$ as the agent is an activated active agent, $pc(Producer) = 1$, $ci(Producer) = [\,]$, and $pv(Producer) = (0)$ as the agent has a single variable $value$ which by code definition is set to 0. Finally, one can write the initial state of the agent $Producer$:

$$S_0(Producer) = (X, 1, [\,], (0))$$

The initial state of the agent Buffer is a 4-tuple, where $am(\mathit{Buffer}) = W$ as the agent is a passive agent, $pc(\mathit{Buffer}) = 0$, $ci(\mathit{Buffer}) = [put]$ as put is the only available procedure, and $pv(\mathit{Buffer}) = (0, \mathit{False})$ as the agent has two variables $value$, and $isFull$ that by code definition are set to 0, and False. Finally, one can write the initial state of the agent Buffer:

$$S_0(\mathit{Buffer}) = (W, 0, [put], (0, \mathit{False}))$$

The initial state of the agent $Consumer$ is the same as the agent $Producer$ with one difference of $am(Consumer) = I$ as the agent is not activated active agent. We can write down the tuple as:

$$S_0(Consumer) = (I, 0, [\ ], (0))$$

3.5 Model dynamics – formal description

In the previous section, the static part of the formal model was discussed, by which it was meant the definition of states of the system. In this section, we will focus on the dynamic part of the model i.e. the rules that govern transitions between the states of the system in the state space of the problem. Here the recipe on how to create explicit reachable states graph (starting from the initial state of the system) is described. The rules interpret the code statements from the code layer deterministically and connections from the graphical layer to create an implicit graph representation of the whole system state space – a formal model of the Alvis. The implicit form of the graph allows creating an explicit form. Two sets of rules specify the implicit form. The first set is called *enabling* rules as for the given state, they define what transitions are allowed from that state, i.e. it defines when a given transition can be executed. The second set of rules is called *firing* rules. Those rules define the result of the execution of given transition, i.e. how the state is modified.

Let us start from the definition of available transitions. The transition is the labelled edge in the state space graph. From the theoretical point of view each statement (except *proc*) is covered by at least one transition with defined *enabling* and *firing* rules. The list of all Alvis transitions for timed α^0 models is given in Table 3.2.

Table 3.2: Transitions

TDelay	a *delay* statement execution
STDelayEnd	termination of agent's suspension
TExec	an *exec* statement execution
TExit	an *exit* statement execution
TIn	initialisation of communication when no peer is available (execution suspension) or reading data from a procedural port – *in* statement
TInAP	calling procedure by an active agent – *in* statement

Transition	Description
STInAP	system version of *TInAP* – for wake up purposes
TInPP	calling procedure by a passive agent – *in* statement
STInPP	system version of *TInPP* – for wake up purposes
TInF	finalisation of a communication with an active agent – *in* statement
STInEnd	abandonment of communication – non-blocking *in* statement
TJump	a *jump* statement execution
TLoop	entering a *loop* statement
TLoopEvery	entering a *loop every* statement
STLoopEnd	termination of current *loop every* run
TNull	*null* statement execution
TOut	initialisation of communication when no peer is available (execution suspension) or writing data to a procedural port – *out* statement
TOutAP	calling procedure by active agent – *out* statement
STOutAP	system version of *TOutAP* – for wake up purposes
TOutPP	calling procedure by a passive agent – *out* statement
STOutPP	system version of *TOutPP* – for wake up purposes
TOutF	finalisation of a communication with an active agent – *out* statement
STOutEnd	abandonment of communication – non-blocking *out* statement
TSelect	entering a *select* statement
TStart	a *start* statement execution
STTime	passage of time

In the non-time version of the language, a subset of the transitions (related to time) is omitted or the passage of time is ignored. The time that is required to finish execution of the instruction in the Alvis model is treated as it would be 0, i.e. virtually ignored. This is in both cases, when the time is explicitly specified in the Alvis code layer (as a parameters of the `delay t`, `loop (every t)`, `in (t) p`, or `out (t) p`), or when the time is specified as the function assigning time required to execute the instruction.

If we want to check whether a transition is enabled, we must always consider it for a given agent and statement. An active agent can execute its statements independently, but a passive agent always works in the context of an active agent. Suppose, an active agent A called a procedure of a passive agent C, namely $C.p$. It means that C works in the context of A. Inside the $C.p$ procedure, a procedure of another passive agent D may be called. Then, both C and D work in the context of A.

To describe the enabling and firing rules, we will use a few auxiliary functions to make the description more concise.

- $context(X)$ – the active agent whose context is used by the passive agent X;
- $caller(X)$ – the agent that called a procedure of X – for the above example we have: $context(C) = caller(C) = A$, $context(D) = A$, and $caller(D) = C$
- $nextpc(n)$ denotes the next program counter determined on the basis of the code structure for the considered agent and current program counter n.

The following assumptions are valid for all **E** rules [171]:

1. A and B are active agents with ports $A.a$ and $B.b$ respectively.
2. C and D are passive agents with ports $C.c$ and $D.d$ respectively.
3. n denotes the index of the statement the considered transition refers to.
4. The underscore mark as a wildcard.
5. For a communication in-out transition the corresponding communication channel is defined in the communication diagram, i.e. the communication described by the analysed transition is formally possible.

Checking if, for example, the $TDelay\ A\ n$ transition is enabled means checking if A can execute a *delay* statement, if $pc(A) = n$.

Rule E1. $TDelay\ A\ n$ is enabled iff: $am(A) = \mathsf{X}$ and $pc(A) = n$ indicates a *delay* statement.

Rule E2. $TDelay\ C\ n$ is enabled iff: $am(C) = \mathsf{T}$ and $pc(C) = n$ indicates a *delay* statement, $am(context(C)) = \mathsf{X}$.

Rule E3. $TExec\ A\ n$ is enabled iff: $am(A) = \mathsf{X}$ and $pc(A) = n$ indicates an *exec* statement.

Rule E4. $TExec\ C\ n$ is enabled iff: $am(C) = \mathsf{T}$ and $pc(C) = n$ indicates an *exec* statement, $am(context(C)) = \mathsf{X}$.

Rule E5. $TExit\ A\ n$ is enabled iff: $am(A) = \mathsf{X}$ and $pc(A) = n$ indicates an *exit* statement.

Rule E6. $TExit\ C\ n$ is enabled iff: $am(C) = \mathsf{T}$ and $pc(C)$ indicates an *exit* statement, $am(context(C)) = \mathsf{X}$.

Rule E7. $TIn\ A.a\ n$ is enabled iff: $am(A) = \mathsf{X}$, $pc(A) = n$ indicates an *in* statement, $ci(A)$ does not contain a $proc$ entry, and transitions $TInAP\ A.a\ \_\ n$ and $TInF\ A.a\ \_\ n$ are not enabled (see rules E9 and E11).

Rule E8. $TIn\ C.c\ n$ is enabled iff:

1. $am(C) = \mathsf{T}$ and $pc(C)$ indicates an *in* statement, $am(context(C)) = \mathsf{X}$ if $C.c$ is a procedure port.
2. $am(C) = \mathsf{T}$, $pc(C)$ indicates an *in* statement, $ci(C)$ does not contain an $proc$ entry, $am(context(C)) = \mathsf{X}$ and transition $TInPP\ C.c\ \_\ n$ is not enabled if $C.c$ is a non-procedure port.

Rule E9. $\mathit{TInAP}\ A.a\ C.c\ n$ is enabled iff: $am(A)$ = X, $pc(A) = n$ indicates an *in* statement, $ci(A)$ does not contain a $proc$ entry, $am(C)$ = W, $ci(C)$ contains $out(C.c)$ entry.

Rule E10. $\mathit{TInPP}\ C.c\ D.d\ n$ is enabled iff: $am(C)$ = T, $pc(C) = n$ indicates an *in* statement, $am(context(C))$ = X, $ci(C)$ does not contain a $proc$ entry, $am(D)$ = W, $ci(D)$ contains $out(D.d)$ entry.

Remark: $C.c$ is non-procedure port.

Rule E11. $\mathit{TInF}\ A.a\ B.b\ n$ is enabled iff: $am(A)$ = X, $pc(A) = n$ indicates an *in* statement, $ci(A)$ does not contain a $proc$ entry, $am(B)$ = W, $ci(B)$ contains $out(B.b)$ entry.

Rule E12. $\mathit{TJump}\ A\ n$ is enabled iff: $am(A)$ = X and $pc(A) = n$ indicates a *jump* statement.

Rule E13. $\mathit{TJump}\ C\ n$ is enabled iff: $am(C)$ = T and $pc(C) = n$ indicates a *jump* statement, $am(context(C))$ = X.

Rule E14. $\mathit{TLoop}\ A\ n$ is enabled iff: $am(A)$ = X and $pc(A) = n$ indicates a *loop* statement.

Rule E15. $\mathit{TLoop}\ C\ n$ is enabled iff: $am(C)$ = T and $pc(C) = n$ indicates a *loop* statement, $am(context(C))$ = X.

Rule E16. $\mathit{TLoopEvery}\ A\ n$ is enabled iff: $am(A)$ = X and $pc(A) = n$ indicates a *loop every* statement.

Rule E17. $\mathit{TLoopEvery}\ C\ n$ is enabled iff: $am(C)$ = T and $pc(C) = n$ indicates a *loop every* statement, $am(context(C))$ = X.

Rule E18. $\mathit{TNull}\ A\ n$ is enabled iff: $am(A)$ = X and $pc(A) = n$ indicates a *null* statement.

Rule E19. $\mathit{TNull}\ A\ n$ is enabled iff: $am(C)$ = T and $pc(C) = n$ indicates a *null* statement, $am(context(C))$ = X.

Rule E20. $\mathit{TOut}\ A.a\ n$ is enabled iff: $am(A)$ = X, $pc(A) = n$ indicates an *out* statement, $ci(A)$ does not contain a $proc$ entry, and transitions $\mathit{TOutAP}\ A.a\ \_\ n$ and $\mathit{TOutF}\ A.a\ \_\ n$ are not enable (see rules E22 and E24).

Rule E21. $\mathit{TOut}\ C.c\ n$ is enabled iff:

1. $am(C)$ = T and $pc(C)$ indicates an *out* statement, $am(context(C))$ = X if $C.c$ is a procedure port.
2. $am(C)$ = T, $pc(C)$ indicates an *out* statement, $ci(C)$ does not contain a $proc$ entry, $am(context(C))$ = X and transition $\mathit{TOutPP}\ C.c\ \_\ n$ is not enabled if $C.c$ is a non-procedure port.

Rule E22. $\mathit{TOutAP}\ A.a\ C.c\ n$ is enabled iff: $am(A)$ = X, $pc(A) = n$ indicates an *out* statement, $ci(A)$ does not contain a $proc$ entry, $am(C)$ = W, $ci(C)$ contains `CIn C_c` entry.

Rule E23. $\mathit{TOutPP}\ C.c\ D.d\ n$ is enabled iff: $am(C)$ = T, $pc(C) = n$ indicates an *out* statement, $am(\mathit{context}(C))$ = X, $ci(C)$ does not contain a $proc$ entry, $am(D)$ = W, $ci(D)$ contains $in(D.d)$ entry. **Remark**: $C.c$ is non-procedure port.

Rule E24. $\mathit{TOutF}\ A.a\ B.b\ n$ is enabled iff: $am(A)$ = X, $pc(A) = n$ indicates an *out* statement, $ci(A)$ does not contain a $proc$ entry, $am(B)$ = W, $ci(B)$ contains $in(B.b)$ entry.

Rule E25. $\mathit{TSelect}\ A\ n$ is enabled iff: $am(A)$ = X and $pc(A) = n$ indicates a *select* statement.

Rule E26. $\mathit{TSelect}\ C\ n$ is enabled iff: $am(C)$ = T and $pc(C) = n$ indicates a *select* statement, $am(\mathit{context}(C))$ = X.

Rule E27. $\mathit{TStart}\ A\ n$ is enabled iff: $am(A)$ = X and $pc(A) = n$ indicates a *start* statement.

Rule E28. $\mathit{TStart}\ C\ n$ is enabled iff: $am(C)$ = T and $pc(C) = n$ indicates a *start* statement, $am(\mathit{context}(C))$ = X.

Rule E29. $\mathit{STInAP}\ A.a\ C.c\ n$ is enabled iff: $am(A)$ = W, $pc(A) = n$ indicates an *in* statement, $ci(A)$ contains $in(A.a)$ entry, $am(C)$ = W, $ci(C)$ contains $out(C.c)$ entry.

Rule E30. $\mathit{STInPP}\ C.c\ D.d\ n$ is enabled iff: $am(C)$ = T, $pc(C) = n$ indicates an *in* statement, $am(\mathit{context}(C))$ = W, $ci(C)$ contains $in(C.c)$ entry, $am(D)$ = W, $ci(D)$ contains $out(D.d)$ entry.
Remark: $C.c$ is non-procedure port.

Rule E31. $\mathit{STOutAP}\ A.a\ C.c\ n$ is enabled iff: $am(A)$ = W, $pc(A) = n$ indicates an *out* statement, $ci(A)$ contains $out(A.a)$ entry, $am(C)$ = W, $ci(C)$ contains $in(C.c)$ entry.

Rule E32. $\mathit{STOutPP}\ C.c\ D.d\ n$ is enabled iff: $am(C)$ = T, $pc(C) = n$ indicates an *out* statement, $am(\mathit{context}(C))$ = W, $ci(C)$ contains $out(C.c)$ entry, $am(D)$ = W, $ci(D)$ contains $in(D.d)$ entry.
Remark: $C.c$ is non-procedure port.

Rule E33. $\mathit{STDelayEnd}\ A\ n$ is enabled iff: $am(A)$ = W, $pc(A) = n$ indicates a *delay* statement, $ci(A)$ contains $\mathit{timeout}(n)$ entry.

Rule E34. $\mathit{STDelayEnd}\ CA\ n$ is enabled iff: $am(C)$ = T, $pc(C) = n$ indicates a *delay* statement, $am(\mathit{context}(C))$ = W, $ci(C)$ contains $\mathit{timeout}(n)$ entry.

Rule E35. $\mathit{STLoopEnd}\ A\ n$ is enabled iff: $am(A)$ = W, $pc(A) = n$ indicates a *loop every* statement, $ci(A)$ contains $\mathit{timeout}(n)$ entry.

Rule E36. $\mathit{STLoopEnd}\ C\ n$ is enabled iff: $am(C)$ = T, $pc(C) = n$ indicates a *loop every* statement, $am(\mathit{context}(C))$ = W, $ci(C)$ contains $\mathit{timeout}(n)$ entry.

Rule E37. $\mathit{STInEnd}\ A\ n$ is enabled iff: $am(A)$ = W, $pc(A) = n$ indicates an *in* statement, $ci(A)$ contains $\mathit{timeout}(n)$ entry.

Rule E38. $\mathit{STInEnd}\ C\ n$ is enabled iff: $am(C)$ = T, $pc(C) = n$ indicates an *in* statement, $am(\mathit{context}(C))$ = W, $ci(C)$ contains $\mathit{timeout}(n)$ entry.

Rule E39. $\mathit{STOutnEnd}\ A\ n$ is enabled iff: $am(A)$ = W, $pc(A) = n$ indicates an *out* statement, $ci(A)$ contains $\mathit{timeout}(n)$ entry.

Rule E40. $\mathit{STOutnEnd}\ C\ n$ is enabled iff: $am(C)$ = T, $pc(C) = n$ indicates an *out* statement, $am(\mathit{context}(C))$ = W, $ci(C)$ contains $\mathit{timeout}(n)$ entry.

Rule E41. $\mathit{STTime}\ d$ – The transition is used to move the clock if none transition is enabled, but at least one transition may be enabled in the future.

The *fire* function is used to compute results of a transition execution. The function takes as its first argument a transition and focuses on results of executing of this simple transition in the given state.

Let us discuss in detail the activity of individual transitions. The following assumptions are valid for all **F** rules:

- A and B are active agents with ports $A.a$ and $B.b$ respectively.
- C and D are passive agents with ports $C.c$ and $D.d$ respectively.
- n denotes the index of the instruction that considered transition refers to.
- $\mathit{context}(C)$ denotes the active agent in which context C works.
- d denotes the number of time-units i.e. the time argument of *delay*, *loop every*, non-blocking *in* and non-blocking *out* statements.
- $\mathit{nextpc}(n)$ denotes the next program counter determined on the basis of the code structure for the considered agent and current program counter n.
- For passive agents, k denotes the current program counter of their context agent.

We will focus on the analysis of these elements that are changed when a transition fires. The results of executing individual transitions in the given state s are as follows:

Rule F1. `TDelay A n` sets $am(A)$ to W and inserts `CTimer n d` entry into $ci(A)$.

Special case: If the `--non-time` option is used, *delay* statements are treated as *null* ones.

Rule F2. `TDelay C n` sets $am(\mathit{context}(C))$ to W and inserts `CTimer n d` entry into $ci(C)$.

Special case: If the `--non-time` option is used, *delay* statements are treated as *null* ones.

Rule F3. `TExec A n` updates value of the corresponding parameter and sets $pc(A)$ to $\mathit{nextpc}(n)$

Special case: If the considered *exec* statement is the agent last statement (i.e. the agent finishes its work after the statement), the transition updates value of the corresponding parameter, sets $pc(A)$ to 0, $am(A)$ to F, and $ci(A)$ to [ ].

Rule F4. `TExec C n` updates value of the corresponding parameter and sets $pc(C)$ to $nextpc(n)$.

Rule F5. `TExit A n` sets $pc(A)$ to 0, $am(A)$ to F, and $ci(A)$ to [ ].

Rule F6. `TExit C n` sets $am(C)$ to W, $pc(C)$ to 0, $ci(C)$ to the set of C procedures accessible in the new state, and

- if the $C.c$ procedure has been called by an active agent A then the `CProc C_c` entry is removed from $ci(A)$ and $pc(A)$ is set to $nextpc(k)$;
- if the $C.c$ procedure has been called by a passive agent D then the `CProc C_c` entry is removed from $ci(D)$ and $pc(D)$ is set to $nextpc(m)$ (where m is the current value of $pc(D)$).

Special case: If the procedure calling was the agent A last statement, the transition sets $pc(A)$ to 0, $am(A)$ to F, and $ci(A)$ to [ ].

Rule F7. `TIn A_a n`

Cases:

1. *Blocking in*: sets $am(A)$ to W and inserts `CIn A_a` entry into $ci(A)$;
2. *Non-blocking in,* $d = 0$: sets $pc(A)$ to $nextpc(n)$;
3. *Non-blocking in,* $d > 0$: sets $am(A)$ to W and inserts `CIn A_a` , `CTimer n d` entries into $ci(A)$

Special cases:

1. If the `--non-time` option is used, all time arguments of a non-blocking *in* statement are treated as zero.
2. If a non-blocking *in* statement with $d = 0$ is the agent last statement, the transition sets $pc(A)$ to 0, $am(A)$ to F, and $ci(A)$ to [ ].

Rule F8. `TIn C_c n`

Cases:

1. $C.c$ is the current procedure port: updates value of the corresponding parameter (if a value has been sent), and sets $pc(C)$ to $nextpc(n)$;
2. $C.c$ is non-procedure port, *blocking in*: sets $am(context(C))$ to W and inserts `CIn C_c` entry into $ci(C)$.
3. $C.c$ is non-procedure port, *non-blocking in,* $d = 0$: sets $pc(C)$ to $nextpc(n)$;
4. $C.c$ is non-procedure port, *non-blocking in,* $d > 0$: sets $am(context(C))$ to W and inserts `CIn C_c` , `CTimer n d` entries into $ci(C)$.

Special case: If the `--non-time` option is used, all time arguments of a non-blocking *in* statement are treated as zero.

Rule F9. `TInAP A_a C_c n` inserts `CProc C_c` into $ci(A)$, sets $am(C)$ to T, sets $pc(C)$ to the index of the first statement in `C_c` procedure, and sets $ci(C)$ to [ ].

Rule F10. `TInPP C_c D_d n` inserts `CProc D_d` into $ci(C)$, sets $am(D)$ to T, sets $pc(D)$ to the index of the first statement in `D_d` procedure, and sets $ci(D)$ to [ ].

Rule F11. `TInF A_a B_b n` updates value of the corresponding parameter of agent A (if a value has been sent), sets $pc(A)$ to $nextpc(n)$, sets $am(B)$ to X, $pc(B)$ to $nextpc(m)$ (where m is the current value of $pc(B)$), and removes `COut B_b`, `CTimer n _` entries from $ci(B)$.

Remark: The $ci(B)$ contains a `CTimer n _` entry only if the communication has been initialised with a non-blocking *out* statement ($d > 0$) and time models are considered (the `--non-time` option is not used).

Special cases:

1. If the *in* statement is the agent A last statement, the transition updates value of the corresponding parameter (if a value has been sent), sets $pc(A)$ to 0, $am(A)$ to F, and $ci(A)$ to [ ].
2. If the *out* statement is the agent B last statement, the transition sets $pc(B)$ to 0, $am(B)$ to F, and $ci(B)$ to [ ].

Rule F12. `TJump A n` sets $pc(A)$ to $nextpc(n)$ (the number of the first statement after the *jump* label).

Rule F13. `TJump C n` sets $pc(C)$ to $nextpc(n)$ (the number of the first statement after the *jump* label).

Rule F14. `TLoop A n` sets $pc(A)$ to $nextpc(n)$ (the number of the first statement in the loop if the corresponding guard is satisfied, or the index of the first statement after the loop otherwise).

Remark: The lack of a guard is treated as the default guard equal to *True*.

Special case: If the guard is not satisfied and the *loop* statement is the agent last statement the transition sets $pc(A)$ to 0, $am(A)$ to F, and $ci(A)$ to [ ].

Rule F15. `TLoop C n` sets $pc(C)$ to $nextpc(n)$.

Rule F16. `TLoopEvery A n` sets $pc(A)$ to $nextpc(n)$ (the number of the first statement in the loop), inserts `CTimer n t` into $ci(A)$, where $t = d -$ *duration of the considered transition*.

Special case: If the `--non-time` option is used, *loop every* statements are treated as general loops with guards equal to *True* (see Rule F14).

Rule F17. `TLoopEvery C n` sets $pc(C)$ to $nextpc(n)$ (the number of the first statement in the loop), inserts `CTimer n t` into $ci(C)$, where $t = d -$ *duration of the considered transition*.

Special case: If the `--non-time` option is used, *loop every* statements are treated as general loops with guards equal to *True* (see Rule F15).

Rule F18. `TNull A n` sets $pc(A)$ to $nextpc(n)$.

Special cases:

1. If the *null* statement is the agent A last statement, the transition sets $pc(A)$ to 0, $am(A)$ to F, and $ci(A)$ to [ ].
2. If the *null* statement is the last one in a *loop every* statement, the transition sets $am(A)$ to W, and $pc(A)$ to the index of the corresponding *loop every* statement.

Rule F19. `TNull C n` sets $pc(C)$ to $nextpc(n)$

Special case: If the *null* statement is the last one in a *loop every* statement, the transition sets $am(context(C))$ to W, and $pc(C)$ to the index of the corresponding *loop every* statement.

Rule F20. `TOut A_a n`

Cases:

1. *Blocking out*: sets $am(A)$ to W and inserts `COut A_a` entry into $ci(A)$;
2. *Non-blocking out,* d = 0: sets $pc(A)$ to $nextpc(n)$;
3. *Non-blocking out,* d > 0: sets $am(A)$ to W and inserts `COut A_a`, `CTimer n d` entries into $ci(A)$

Special cases:

1. If the `--non-time` option is used, all time arguments of a non-blocking *out* statement are treated as zero.
2. If a non-blocking *out* statement with d = 0 is the agent last statement, the transition sets $pc(A)$ to 0, $am(A)$ to F, and $ci(A)$ to [ ].

Rule F21. `TOut C_c n`

Cases:

1. $C.c$ is the current procedure port: sets $pc(C)$ to $nextpc(n)$;
2. $C.c$ is non-procedure port, *blocking out*: sets $am(context(C))$ to W and inserts `COut C_c` entry into $ci(C)$.
3. $C.c$ is non-procedure port, *non-blocking out,* d = 0: sets $pc(C)$ to $nextpc(n)$;
4. $C.c$ is non-procedure port, *non-blocking out,* d > 0: sets $am(context(C))$ to W and inserts `COut` $\hookrightarrow$ `C_c`, `CTimer n d` entries into $ci(C)$.

Special case: If the `--non-time` option is used, all time arguments of a non-blocking *out* statement are treated as zero.

Rule F22. `TOutAP A_a C_c n` inserts `CProc C_c` into $ci(A)$, sets $am(C)$ to T, sets $pc(C)$ to the index of the first statement in `C_c` procedure, and sets $ci(C)$ to [ ].

Rule F23. `TOutPP C_c D_d n` inserts `CProc D_d` into $ci(C)$, sets $am(D)$ to T, sets $pc(D)$ to the index of the first statement in `D_d` procedure, and sets $ci(D)$ to [ ].

Rule F24. `TOutF A_a B_b n` sets $pc(A)$ to $nextpc(n)$, updates value of the corresponding parameter of agent B (if a value has been sent), sets $am(B)$ to X, sets $pc(B)$ to $nextpc(m)$ (where m is the current value of $pc(B)$), and removes the `CIn B_b`, `CTimer n _` entries from $ci(B)$.

Remark: The $ci(B)$ contains a `CTimer n _` entry only if the communication has been initialised with a non-blocking *in* statement ($d > 0$) and time models are considered (the `--non-time` option is not used).

Special cases:

1. If the *out* statement is the agent A last statement, the transition sets $pc(A)$ to 0, $am(A)$ to F, and $ci(A)$ to [ ].
2. If the *in* statement is the agent B last statement, the transition updates the value of the corresponding parameter (if a value has been sent), sets $pc(B)$ to 0, $am(B)$ to F, and $ci(B)$ to [ ].

Rule F25. `TSelect A n` sets $pc(A)$ to $nextpc(n)$ (the number of the first statement in the first open branch, or the index of the first statement after the *select* statement if all branches are closed).

Special case: If all branches are closed and the *select* statement is the agent last statement, the transition sets $pc(A)$ to 0, $am(A)$ to F, and $ci(A)$ to [ ].

Rule F26. `TSelect C n` sets $pc(C)$ to $nextpc(n)$ (the number of the first statement in the first open branch, or the index of the first statement after the *select* statement if all branches are closed).

Rule F27. `TStart A n` sets $pc(A)$ to $nextpc(n)$, and if the agent B (parameter of the *start* statement) is in the *init* mode sets $am(B)$ to X and $pc(B)$ to 1.

Special case: If the considered *start* statement is the agent last statement, the transition sets $pc(A)$ to 0, $am(A)$ to F, and $ci(A)$ to [ ].

Rule F28. `TStart C n` sets $pc(C)$ to $nextpc(n)$, and if the agent B (parameter of the *start* statement) is in the *init* mode sets $am(B)$ to X and $pc(B)$ to 1.

Rule F29. `STInAP A_a C_c n` sets $am(A)$ to X, removes `CIn A_a`, `CTimer n _` entries from $ci(A)$, inserts `CProc C_c` into $ci(A)$, sets $am(C)$ to T, sets $pc(C)$ to the index of the first statement in the `C_c` procedure, and sets $ci(C)$ to [ ].

Remark: The $ci(A)$ contains a `CTimer n _` entry only if the communication has been initialised with a non-blocking *in* statement ($d > 0$) and time models are considered (the `--non-time` option is not used).

Rule F30. `STInPP C_c D_d n` sets $am(context(C))$ to X, removes `CIn C_c`, `CTimer n _` entries from $ci(C)$, inserts `CProc D_d` into $ci(C)$, sets $am(D)$ to T, sets $pc(D)$ to the index of the first statement in the `D_d` procedure, and sets $ci(D)$ to [ ].

Remark: The $ci(C)$ contains a `CTimer n _` entry only if the communication has been initialised with a non-blocking *in* statement ($d > 0$) and time models are considered (the `--non-time` option is not used).

Rule F31. `STOutAP A_a C_c n` sets $am(A)$ to X, removes `COut A_a`, `CTimer n _` entries from $ci(A)$, inserts `CProc C_c` into $ci(A)$, sets $am(C)$ to T, sets $pc(C)$ to the index of the first statement in the `C_c` procedure, and sets $ci(C)$ to [ ].

Remark: The $ci(A)$ contains a `CTimer n _` entry only if the communication has been initialised with a non-blocking *out* statement ($d > 0$) and time models are considered (the `--non-time` option is not used).

Rule F32. `STOutPP C_c D_d n` sets $am(context(C))$ to X, removes `COut C_c`, `CTimer n _` entries from $ci(C)$, inserts `CProc D_d` into $ci(C)$, sets $am(D)$ to T, sets $pc(D)$ to the index of the first statement in the `D_d` procedure, and sets $ci(D)$ to [ ].

Remark: The $ci(C)$ contains a `CTimer n _` entry only if the communication has been initialised with a non-blocking *out* statement ($d > 0$) and time models are considered (the `--non-time` option is not used).

Rule F33. `STDelayEnd A n` sets $am(A)$ to X, sets $pc(A)$ to $nextpc(n)$, removes `CTimeout n` from $ci(A)$.

Special case: If the considered *delay* statement is the agent last statement the transition sets $pc(A)$ to 0, $am(A)$ to F, and $ci(A)$ to [ ].

Rule F34. `STDelayEnd C n` sets $am(context(C))$ to X, sets $pc(C)$ to $nextpc(n)$, removes `CTimeout n` from $ci(C)$.

Rule F35. `STLoopEnd A n` sets $am(A)$ to X, removes `CTimeout n` from $ci(A)$.

Rule F36. `STLoopEnd C n` sets $am(context(C))$ to X, removes `CTimeout n` from $ci(C)$.

Rule F37. `STInEnd A n` sets $am(A)$ to X, sets $pc(A)$ to $nextpc(n)$, removes `CTimeout n` and `CIn` $\hookrightarrow$ `A_a` entries from $ci(A)$.

Special case: If the considered *in* statement is the agent last statement, the transition sets $pc(A)$ to 0, $am(A)$ to F, and $ci(A)$ to [ ].

Rule F38. `STInEnd C n` sets $am(context(C))$ to X, sets $pc(C)$ to $nextpc(n)$, removes `CTimeout n` and `CIn C_c` entries from $ci(C)$.

Rule F39. `STOutEnd A n` sets $am(A)$ to X, sets $pc(A)$ to $nextpc(n)$, removes `CTimeout n` and `COut A_a` entries from $ci(A)$.

Special case: If the considered *out* statement is the agent last statement, the transition sets $pc(A)$ to 0, $am(A)$ to F, and $ci(A)$ to [ ].

Rule F40. `STOutEnd C n` sets $am(context(C))$ to X, sets $pc(C)$ to $nextpc(n)$, removes `CTimeout` ↪ `n` and `COut C_c` entries from $ci(C)$.

Rule F41. `STTime d` The transition represent a passage of time. It updates time entries of all agents.

As an example of the application of **E** and **F** rules, we can again consider the Producer–Consumer problem 1.9. For start, we can consider the initial state of that problem shown in Sec. 3.4. As a reminder the initial state in a more concise way is:

$$S_0 = ((X, 1, [\,], (0)), (W, 0, [put], (0, False)), (I, 0, [\,], (0)))$$

By analysing all rules from **R** set, we can conclude that in that state only E14 is applicable. This is because $am(Producer) = X$ and $pc(Producer) = 1$, therefore, *Producer* agent is in executing mode and its program counter indicates unconditional `loop` instruction. As a result the transition `TLoop Producer 1` is enabled in the initial state. As we have only one enabled transition, we can therefore focus on **F** rules set. For that, transition F14 can fire. As a result of firing, the transition `TLoop Producer 1` according to rule F14, the new state of the system is as follows:

$$S_0 = ((X, 2, [\,], (0)), (W, 0, [put], (0, False)), (I, 0, [\,], (0)))$$

This is because $nextpc(1) = 2$ as loop instruction has no condition.

Secondly, let us consider the following a little bit more interesting state (not reachable from previous one as *Consumer* agent is not possible to start, but it would be reachable if *Consumer* agent would be active in the inital state of the whole system):

$$S_0 = ((X, 3, [\,], (0)), (W, 0, [put], (0, False)), (X, 2, [\,], (0)))$$

In that specific state there are two rules that are applicable: E22 rule as $am(Producer) = X$ and $pc(Producer) = 3$ and therefore program counter indicates `out` instruction, as $ci(Producer) = [\,]$ (is empty), $am(Buffer) = W$, and finally $ci(Buffer) = [put]$ and E7 rule as rules E9 and E11 for *Consumer* agent are not active. The E9 is not enabled as there is no entry *get* in the $ci(Buffer)$, and E11 is not enabled as there is no active agent connected to the *pull* port of *Consumer*. Therefore, the transitions: `TOutAP` ↪ `Producer.push Buffer.put 3` and `TIn Consumer 2` are enabled in that state.

An execution of the **`in`** and **`out`** instructions may affect more than one agent in the system. This is because those instructions are related to exchanging messages or signals between agents, and therefore it usually happens that states of two agents are affected. If several agents want to establish communication with each other simultaneously, some conflicts among agents can surface. The Alvis language allows assignment of ten levels of priorities to any agent in the system to give the modeller the possibility to resolve some conflicts in a deterministic way. Those priorities are between 0 and 9, where 0 means the highest priority and 9 means the lowest priority. The conflicts are resolved dynamically by the algorithm while computing the LTS. Therefore, the priorities does not influence the Alvis compiler behaviour.

As a simple example of the conflict, we could imagine an output procedure $read$ of the passive agent `Sensor`. The procedure is shared between two active agents `ImportantTask`, `InsignificantTask` ↪ . Both active agents could request their **`in`** instructions to access the procedure `read` in the infinite loop. If both of them execute their communication instructions due to mutual exclusion mechanism, only one of them will get access. The "loser" will have to wait until the procedure is available again. Without priorities assigned, the outcome is indeterministic, and every agent has an equal probability of accessing the procedure. By assigning higher priority (lower number) to the `ImportantTask`, a designer can get rid of the indeterminism in the system.

3.6 LTS and non-timed Alvis

States and transitions of an Alvis model are linked together into a graph like structure which is a *Labelled Transition System* (LTS). An LTS is used for verifying the corresponding model using model checking techniques [10], [38]. Alvis models can be verified using popular model checkers like nuXmv [30], [20] and CADP [69], [60].

In this section, we will consider the non-timed version of the Alvis model interpretation. In that case, we analyse the model on the level of the execution of a single transition at the time. Then all the possible execution of the model is represented as a directed graph [170]. The nodes of such a graph are the possible states of the system, and arcs represent transitions. A single arc corresponds to a single transition of the system made by just one agent at a time (there can be two agents involved only in the case when the instruction affects two agents at the same time, e.g. communication instructions, or start instruction).

Definition 3.6.1. *Labelled transition system* (in short LTS graph or just LTS) is the 4-tuple $LTS = (S, A, \rightarrow, s_0)$, where:

- S is the set of states,
- A is the set of actions,
- $\rightarrow \subseteq S \times A \times S$ is the transition relation,
- $S_0 \in S$ is initial state.

Definition 3.6.2. Let an Alvis model $\mathbf{A} = (D, B, \alpha^0)$ be given. A state S' is **directly reachable** from a state S iff $\exists_{t \in \mathcal{T}} : S - t \to S'$. A state S' is **reachable** from a state S iff there exists sequence of directly reachable states $S^1, \ldots, S^k$ by the sequence of transitions $t^1, \ldots, t^k \in \mathcal{T}$ such that $S = S^1 - t^1 \to S^2 - t^2 \to \ldots - t^k \to S^{k+1} = S'$

By the symbol $\mathcal{R}(S_0)$ we designate the set of all reachable states of the Alvis model from the initial state S_0

Then in the context of the Alvis model, we will narrow the general definition of the LTS to the 4-tuple of $LTS = (\mathcal{R}(S_0), \mathcal{T}, \to, S_0)$, where $\to = \{(S, t, S') : S - t \to S' \wedge S, S' \in \mathcal{R}(S_0)\}$

Chapter 4

Intermediate Haskell Representation of Alvis model

The Intermediate Haskell Representation (abbreviated IHR) represents an Alvis model in Haskell code as an implicit LTS graph. The IHR is a code of a standalone application. The code can be arbitrarily but only optionally modified, or compiled and run to generate the explicit LTS graph representation. By explicit form of the LTS graph, it is understood one of the two most important representations of a graph, namely, adjacency matrix nor adjacency list [48, 158]. Thanks to the IHR being represented in a general-purpose programming language, namely Haskell [142, 143, 52, 116], the representation is exceptionally flexible, and various extensions can be easily added. The extensions can support, for example, different system layers, various LTS graph generation algorithms, provide or extend the Alvis model to support new modelled domain more seamlessly, or provide a converter to new format of explicit graph representation. From this chapter, one can learn how information from code layer and graphical layer are translated into IHR. The algorithms that realise requirements presented in the previous chapter are described below.

The implicit representation of an LTS graph is based on two functions `enable` and `fire` that are discussed in detail in Sec. 4.3, and Sec. 4.4 respectively. The Haskell representation of the behaviour of agents of the modelled system translates directly to the specification of the functions `enable` and `fire` ↪ . The compiler generates `enable` and `fire` methods definitions for every model. Combining results of those functions with the initial state of the system is a full recipe for building an explicit representation of the graph. The nodes or vertices of the LTS graph are particular system states. The edges of the LTS graph represent transitions. Function `enable` calculates transitions enabled in the given system state constrained to particular agent transitions and function `fire` calculates new system state based on previous system state and a transition enabled in that state. The function `enable` type signature is as presented in Listing 4.1.

```
enable :: State −> Agent −> [TTransition]
```

Listing 4.1: Type signature of the function enable

The function `fire` calculates the directly reachable states for the given system state and the given transition. In most of the cases, there is only a single state that is directly reachable from the particular system state through the particular transition – representing a statement like, e.g. **`null`**, **`start`**, and others. There is only a single exception to the transition related to the exec statement with a special function `pick` that introduce non-deterministic behaviour in the system. The function `fire` type signature is presented in Listing 4.2.

```
fire :: TTransition -> State -> [State]
```

Listing 4.2: Type signature of the function fire

Therefore, before introduction of the functions, the detailed representation of the static system state in the IHR is discussed in Sec. 4.1. Next, the detailed representation of transitions in the IHR that model the dynamics of the system is discussed in Sec. 4.2.

4.1 Fundamental definitions of model's data

All necessary data structures that represent the state of the system as a whole and the parts that the system's state is composed of is discussed in the following sections. The representation of the system state is introduced in a top-down manner beginning from the definition of the whole state of the system in Sec. 4.1.1. All necessary symbols are then introduced in the following sections in order of appearance, so we can see the context first and then understand the details.

4.1.1 System state

The Haskell model of a system state is generated in the compilation process as a dedicated type for the given Alvis model. Different Alvis models may have different IHR state representation. The type depends on the number of agents in the non-hierarchical Alvis model (cf. Def. 3.1.1), their names, and order of an agent's behavior definitions in the code layer (cf. Sec. 3.2).

In the current version of the compiler, a state of the system is designed as a type alias [121, 116] of the tuple of particular agent's state types (cf. Sec. 4.1.2). A general scheme of the generated IHR representation of the system's state is presented in Listing 4.3.

```
type State = (FirstAgentState, SecondAgentState, ThirdAgentState, ...)
```

Listing 4.3: Template of generated code for the type State. The names of FirstAgent, SecondAgent, ThirdAgent are suffixed with State in order to represent state of particular agent.

The name of the type alias for system state is the same for every model – `State`. This name is used in the type definition signatures of the helper functions, which will be explained in this chapter's following sections. What changes for different models is the definition of the system state, i.e. the tuple representing states of particular agents.

In the example of the Producer-Consumer problem (cf. Sec. 1.9), the compiler generates the system state type definition that is presented in Listing 4.4.

```haskell
type State = (ProducerState, BufferState, ConsumerState)
```

Listing 4.4: Example of the generated code for type `State` of the Producer-Consumer problem.

4.1.2 Agent state

The Alvis compiler generates dedicated type aliases for every agent in the modelled system to represent their states. When agents share the common behaviour with another agent (cf. Sec. 3.2), there is generated a dedicated entry for every agent anyway. There is no need to differentiate the passive agent state from the active agent state. Therefore, both states are represented as the same type in the Haskell model. Moreover, for the simplicity, Alvis compiler uses simple and naïve name mangling algorithm, which adds a suffix `State` to the name of the agent. This algorithm has an advantage of improving the readability of the IHR code because the names in the Haskell model corresponds in a straightforward way to the original agents as specified by the user. The main disadvantage of this approach is that this, together with the agent's type, can lead to subtle errors when the names of the agent in the modelled system, will follow the pattern: there is at least one agent whose name is `XYZ` and the other agent whose name is `XYZState`. However, the problem is not silently ignored, and the later compilation process will fail for the IHR model. The problem could be avoided though, by enriching the name-mangling algorithm by, e.g., conditional addition of consequent numbers and avoiding further collisions, but it will make the correspondence between Alvis model and IHR representations less visible. A general scheme of the generated Haskell code is provided in Listing 4.5.

```haskell
type FirstAgentState = (Mode, Int, Set ContextInfo,
  (Var1Type, Var2Type, Var3Type, ...))
type SecondAgentState = (Mode, Int, Set ContextInfo,
  (Var1Type, Var2Type, Var3Type, ...))
type ThirdAgentState = (Mode, Int, Set ContextInfo,
  (Var1Type, Var2Type, Var3Type, ...))
...
```

Listing 4.5: Template of generated code for types *`Agent`***`State`**

The type alias name is the agent's name from the Alvis model, appended with a suffix `State` for every agent in the system. The agent state is a 4-tuple that consists of the mode of the agent (discussed in Sec. 4.1.3), the program counter, that is represented as a Haskell built-in type **`Int`**, the third element of the tuple is context information list (discussed in Sec. 4.1.5), and finally, the last element is a tuple representing all variables of the related agent. The tuple of the variables of an agent can be as simple as `()` – the unit type – a zero tuple, in the case of the agent with no variables or arbitrarily complicated tuple otherwise. The variable tuple is discussed in Sec. 4.1.7.

For the example of the Producer-Consumer problem, cf. Sec. 1.9, the compiler generates the following types 4.6.

```
type ProducerState = (Mode, Int, Set ContextInfo, (Int))
type BufferState = (Mode, Int, Set ContextInfo, (Int, Bool))
type ConsumerState = (Mode, Int, Set ContextInfo, (Int))
```

Listing 4.6: Example of the generated code representing state types for agents of Producer-Consumer problem

As more complicated example one can consider the following Alvis code layer (Listing 4.7).

```
data RemoteState = NotConnected | Connected | Establishing

agent A, B {
  message :: String = "hello";
  signals :: [Int] = [0,0,0,0,0];
  remoteState :: RemoteState = NotConnected;
  exit;
}

agent C {
  remoteState :: RemoteState = NotConnected;
  exit;
}
```

Listing 4.7: Example of the model of three active agents with up-to three variables and the user defined type RemoteState

For the system with the code layer presented in Listing 4.7, the compiler generates the following agent type definitions, cf. Listing 4.8.

```
type AState = (Mode, Int, Set ContextInfo,
   (String, [Int], RemoteState))
type BState = (Mode, Int, Set ContextInfo,
   (String, [Int], RemoteState))
type CState = (Mode, Int, Set ContextInfo,
   (RemoteState))
```

Listing 4.8: Example of the generated code representing state types of agents for problem from Listing 4.7

4.1.3 Agent's mode

The agent's mode of the Alvis model (cf. Sec. 3.4) is represented as a new algebraic type, which in Haskell, it is introduced with the keyword **`data`**[121, 143, 52]. This approach allows to create values of the agent mode, i.e., to set or test the mode of any agent state in a Haskell model, in a way that visibly corresponds to their theoretically possible states by using the short name of the modes. That is more readable approach than using just integer type instead. This IHR definition is model independent, and it stays the same across different models. There is no distinction between modes of a passive agent and an active agent. Therefore, e.g. *waiting* mode of a passive agent and *waiting* mode of an active agent are both represented as the value `W`. The generated type (cf. Listing 4.9) derives from built-in Haskell classes of **`Eq`**, **`Ord`**, and **`Show`**. Therefore, the value `Mode` can be tested whether they are the same or different from each other (instance of **`Eq`**), they can be ordered deterministically (instance of **`Ord`**), and therefore, they can be put into, e.g. a tree based implementation of the set. Finally, one can transform the value of `Mode` into its textual representation (instance of **`Show`**). What is more important, the derived types such as *`Agent`*`State` (cf. Sec. 4.1.2) or `State` (cf. Sec. 4.1.1) are having the same properties as well.

```haskell
data Mode
  = F -- ^ Finished
  | I -- ^ Init
  | R -- ^ Ready
  | T -- ^ Taken
  | W -- ^ Waiting
  | X -- ^ running (eXecuting)
  deriving (Eq, Ord, Show)
```

Listing 4.9: Mode data type definition

4.1.4 Program counter

Program counter value of the Alvis model, cf. Sec. 3.4, is represented as a simple Haskell built-in **`Int`** in the IHR. The compiler scans all instructions of a specific agent from its code layer 3.2, and it assigns a value of positive integer (starting from 1) to every instruction according to their position in the source code. Then the compiler can generate tests of the program counter values in different states, e.g. in the `enable` function 4.3. The value of an agent's program counter is modified by the function `fire` 4.4.

4.1.5 Context info

A context information of an Alvis agent (cf. Sec. 3.4) is represented as a `Set` of `ContextInfo` data type. The set is used for the performance reason instead of the standard linked list data structure. The Haskell, parametric container type `Set` implements a set of elements of the type provided by its parameter using a balanced tree data structure [109, 2]. Therefore, the implementation of the set requires their elements to be orderable. The parameter type `ContextInfo` is generated by the compiler, cf. Listing 4.10. The generated code is model-independent. It provides additional information that describes the current state of the agent. E.g., context info describes the port on which an active agent waits to complete its communication instruction. Some context information is restricted only to passive agents, and some of them are bound to active agents only, as was already discussed in Sec. 3.4.

```
data ContextInfo
  = CProc Port
  | CIn Port
  | COut Port
  | CTimer Int Int
  | CSFT Int
  | CNSFT Int
  | CLock
  | CNone
  | COTimer Int Int
  deriving (Eq, Ord)
```

Listing 4.10: ContextInfo data type definition

Most of the value constructors of the data type `ContextInfo` have additional fields. Here the first string literal after the sign '|' is the name of the value constructor itself, and the second, and further string literals are the types of the first and further fields of that particular value. The type `Port` related to some the value constructors is discussed further in details in Sec. 4.1.6. The meaning of the value constructors was already discussed in the Sec. 3.4, below only new artificial values are discussed:

- `CProc Port` – represents $proc(p)$;
- `CIn Port` – represents $in(p)$;
- `COut Port` – represents $out(p)$;
- `CTimer` **`Int Int`** – represents $timer(n, t)$;
- `CTimeout` **`Int`** – represents $timeout(n)$;
- `CSFT` **`Int`** – represents $sft(t)$;
- `CNSFT` **`Int`** – This value is a transient one and is used only temporarily as a result of the computation of next multistep. It never occurs in the states of LTS graph nodes;
- `CLock` – represents $lock$;
- `CNone` – This value is transient, and it is used only as a result of searching functions to denote that none a result was found. Likewise, the CNSFT, the value never appears in the final LTS graph's nodes;
- `COTimer` **`Int Int`** – This value is transient, and it is used only temporarily as a result of the computation of a next multistep. It never occurs in the states of LTS graph nodes. The first parameter of the **`Int`** type is the number of the statement and the second parameter is the number of time units

required to finish the current statement.

The data type `ContextInfo` is equipped with the customized **`show`** method (shown in Listing 4.11) instead of automatically generated **`Show`** instance by the Haskell compiler.

```
instance Show ContextInfo where
  show (CProc a) = "proc(" ++ (show a) ++ ")"
  show (CIn a) = "in(" ++ (portName (show a)) ++ ")"
  show (COut a) = "out(" ++ (portName (show a)) ++ ")"
  show (CTimer n d) = "timer(" ++ (show n) ++ "," ++ (show d) ++ ")"
  show (CTimeout n) = "timeout(" ++ (show n) ++ ")"
  show (CSFT d) = "sft(" ++ (show d) ++ ")"
  show (CLock a) = "lock(" ++ (show a) ++ ")"
  show _ = ""
```

Listing 4.11: ContextInfo instance of the class ***Show***

4.1.6 Port

All ports of the Alvis model (defined in the communication diagram, cf. Sec. 3.1) are represented as a single Haskell data type in the IHR. The naming convention of the value constructors in the IHR replaces the dot from Alvis theoretical model with an underscore because the dot is restricted scope operator in Haskell and cannot be used as a part of the name of the value constructor, cf. Listing 4.12. The compiler adds unique value at the end of the list, namely `None` which is restricted for use in searching functions to indicate no result. The value `None` never appears in the states of the system.

```
data Port
  = FirstAgentName_ItsFirstPortName
  | FirstAgentName_ItsSecondPortName
  | SecondAgentName_ItsFirstPortName
  | SecondAgentName_ItsSecondPortName
  | ThirdAgentName_ItsFirstPortName
  | None
  deriving (Eq,Ord)
```

Listing 4.12: Port ***data*** type

For the example of the Producer-Consumer problem (cf. Sec. 1.9), the compiler generates a definition of the type `Port` as presented in Listing 4.13.

```
data Port
  = Producer_push
  | Buffer_get
  | Buffer_put
  | Consumer_pull
  | None
  deriving (Eq,Ord)
```

Listing 4.13: Example of the generated code for the type `Port` for the Producer-Consumer problem

Compiler generates dedicated **`Show`** instance for the type `Port`

4.1.7 Variables tuple

All variables of a specific agent of the Alvis model are represented as a dedicated tuple of all variables in the IHR. This approach takes advantage of the constraint that all variables have to be predeclared in the Alvis code layer. For each agent in a model, the compiler generates a tuple signature that can represent all variables of that agent. The order of variables inside the generated tuple is the same as in the code layer. As an example, please confront the code layer presented in Listing 4.14 with a corresponding Haskell representation of all variables tuple (Listing 4.15), and the complete state of that agent, that would be generated by the compiler in that case (Listing 4.16).

```
agent A {
 var1 :: Type1 = initialValue1;
 var2 :: Type2 = initialValue2;
 var3 :: Type3 = initialValue3;

 exit;
}
```

Listing 4.14: Example of the agent A code – Alvis code layer

```
(Type1, Type2, Type3)
```

Listing 4.15: Generated tuple signature that represents types of all variables of the agent A

```
data AState = (Mode, Int, Set ContextInfo, (Type1, Type2, Type3))
```

Listing 4.16: Generated dedicated *data* type of the agent A state composed of the tuple representing variables of the agent A.

Representation of all variables as a tuple has some limitations such that, the modeller is responsible for providing its instances of classes Read, Eq, Show, and create a dedicated tuple type when the number of variables is larger than 15. The description of a computation that has to be performed on the variables of an agent is discussed in together with `fire` function Sec. 4.4. Only execution of transitions related to the instructions: **`exec`**, **`loop`**, **`select`**, **`in`**, **`out`**, and guarded procedures may access the agent local variables. And only **`exec`**, and **`in`** may modify a single local agent variable per instruction.

Currently, as it was mentioned previously, the Intermediate Haskell Representation of agent's variables

The agents' variables definition in the IHR is easily extendable from the case of statically predefined variables to the case of dynamically allocated variables. The dynamically allocated variables naturally emerge if one would extend a single block of code of active agent's behaviour specification with the possibility of usage of an arbitrary number of callable functions within an agent's block of code. Then the dynamically allocated variables appear within the context of any function as a function of local variables. In most imperative languages, like e.g., C or Java, a function local variable is allocated on the stack, and a global variable is allocated in the static global memory storage once the program is launched. For those familiar with C language, one can change the default allocation policy of the C compiler for a particular variable with C keywords. The allocation on the stack is default, but can be explicitly stated by a keyword `auto`. The global memory allocation can be achieved by a keyword `static`. Allocation on the stack has several advantages. Firstly, a variable is automatically freed once the execution of the function is completed. Secondly, every call to the function (even recursive one) has its copy of the variable that does not interfere with instances of the variable from the previous call. The IHR model of agent's variable could be therefore extended by adding a stack-like data structure to the agent's variable tuple as an additional field. Then every function invocation pushes a new frame with the function's variables onto a stack-like data structure, and every return from the function pops the top frame from that stack. Any calculation involving variables may be performed on variables from the global memory pool and variables from the topmost frame on the stack.

4.1.8 Helper methods

The IHR was equipped with a set of helper functions that simplifies extraction of particular elements from a whole system state. They enable querying the system state, or they provide additional information on the system as a whole.

The list of all agent types is held in the constant `agents`. The general template is shown in Listing 4.17. Moreover, `agents` example as generated for the Producer-Consumer problem, cf. Sec. 1.9, is shown in Listing 4.18. The list contains agent types (their names) in the same order as their definition order in code layer, cf. Sec. 3.2. The type of `Agent` is discussed in detail in Sec. 4.1.9.

```
agents :: [Agent]
agents = [FirstAgent, SecondAgent, ThirdAgent, ...]
```

Listing 4.17: Template code of agents constant in the IHR

```
agents :: [Agent]
agents = [Producer, Buffer, Consumer]
```

Listing 4.18: Example value of agents constant in the Producer-Consumer problem

A constant representing the number of all agents, both active and passive in the system is called `modelSize`. The integer value representing the number is generated by the compiler per model and inserted in the place of `n` symbol in the template code from Listing 4.19. The template applied to the Producer-Consumer problem (Sec. 1.9) is shown in Listing 4.20.

```
modelSize :: Int
modelSize = n
```

Listing 4.19: Constant modelSize – represents number of agents in the system

```
modelSize :: Int
modelSize = 3
```

Listing 4.20: Example value of the constant modelSize for the Producer-Consumer problem

A function of `agentNumber` converts value of type `Agent` to its position index in the `agents` list constant and the position of the corresponding agent state within the system state that is described in Sec. 4.1.1. The position value of the agent in the system state, from now on, will be designated as **agent index**. The agent index is required by several helper functions to query directly for a specific agent state value. The function calculates position dynamically based on the `agents` value, and its implementation is the same for different models.

```
agentNumber :: Agent -> Int
```

Listing 4.21: Function agentNumber – returns the position index, **agent index**, for the given agent

A function of `agentName` converts value of type `Port` to the value of type `Agent`. That means that for any port in the system, the function returns the agent that is the owner of the given port. Every port in the system belongs to the precisely one agent in the system. The template used by the compiler is shown in Listing 4.22. The application of the template to the Producer-Consumer problem is shown in Listing 4.23.

```
agentName :: Port -> Agent
agentName :: FirstAgent_port1 = FirstAgent
agentName :: FirstAgent_port2 = FirstAgent
agentName :: SecondAgent_port1 = SecondAgent
```

Listing 4.22: Function agentName – returns the agent that is owner of the given port

```
agentName :: Port -> Agent
agentName Producer_push = Producer
agentName Buffer_get = Buffer
agentName Buffer_put = Buffer
agentName Consumer_pull = Consumer
agentName None = error "None␣port␣has␣no␣assigned␣agent"
```

Listing 4.23: Example of the generated function agentName for the Producer-Consumer problem

The following functions of `takeam`, `takepc`, and `takeci` return the agent's mode, the value of its program counter, or the agent's context information list, correspondingly, as one can investigate in Listing 4.24. The functions require the system state value (cf. Sec. 4.1.1) to be provided, and the agent index, (cf. Listing 4.21). The returned values are tested or updated by the functions `enable` and `fire`. There is no special function to extract particular variable nor the tuple of all variables because any modification or test of variables is only within the context of the functions `fire`, `enable`, **`init`** where the access to any variable is provided by the Haskell pattern matching mechanism.

```
takeam :: Int -> State -> Mode
takepc :: Int -> State -> Int
takeci :: Int -> State -> Set ContextInfo
```

Listing 4.24: Signature of functions takeam, takepc, takeci that extract mode, program counter, or context information list for the given agent index and from the given system state

The modification of a given agent's context list or all context lists of all agents at once is simplified by the functions `updateContextInfo` and `updateAllContextInfo`. When the modification of a specific agent's context list is required, the former function is used with arguments of the agent index, a function that updates the context list, and a state of the system for which the change has to occur. The function returns the updated system state. When the same modification of context list for all agents in the system is required there is `updateAllContextInfo` method that simplifies the modification.

```
updateContextInfo :: Int -> (Set ContextInfo -> Set ContextInfo) -> State -> State
updateAllContextInfo :: (Set ContextInfo -> Set ContextInfo) -> State -> State
```

Listing 4.25: Type signature of functions updateContextInfo and updateAllContextInfo update context information list of a specific agent or all agents at once

For example, the following function of `addContextInfo` is simplified with the above defined function `updateContextInfo`. Here, additionally to the type signature of the function, there is presented the implementation of that function to show how it benefits from using the function `updateContextInfo`. The implementation passes a function `Set.`**`insert`** as the second argument to the function `updateContextInfo`. `Set.`**`insert`** takes two arguments, but here just one value of type `ContextInfo` is applied, and the second argument is left not applied. The result is a new function that takes a single argument of a context list to be modified [86].

```
addContextInfo :: Int -> ContextInfo -> State -> State
addContextInfo n info state = updateContextInfo n (insert info) state
```

Listing 4.26: Type signature of function addContextInfo and its implementation

The function of `isProc` tests if its only argument is the value of `CProc`. This test is used by two functions `findProc`, and `procFree`.

```
isProc :: ContextInfo -> Bool
```

Listing 4.27: Type signature of function isProc tests if ContextInfo value is of instance of CProc

The function `procAgent` converts a value of type `ContextInfo` to the value of type `Agent`. The conversion is not always possible. It is restricted only to the values created with the value constructor `CProc` ↪ . The implementation returns, for any value `CProc`, an agent that is the owner of the procedural port extracted from value of `ContextInfo`.

```
procAgent :: ContextInfo -> Agent
```

Listing 4.28: Type signature of function procAgent returns owner agent of the given value CProc

The function `procFree` tests whether the given context list does not include any entry created by the value constructor `CProc`. The presence of an instance of `CProc` signifies that the given agent is performing a procedure of another passive agent, cf. Sec. 3.4.

```
procFree :: Set ContextInfo -> Bool
```

Listing 4.29: Type signature of function procFree tests if context list is free of procedural calls

The function of `findProc` searches through the given context list for the first occurrence of `CProc` instance related to the given agent. If there is no such entry the dummy value of `CProc None` is returned.

```
findProc :: Agent -> Set ContextInfo -> ContextInfo
```

Listing 4.30: Type signature of function findProc returns called procedure by the given agent

4.1.9 Agent type

The compiler generates a new `Agent` algebraic data type representing all agents' names in the modelled system. The values of `Agent` data type are used in the `enable` function (Sec. 4.3) to distinguish for which agent transitions are tested. The value of `Agent` is transformed to agent index by means of `agentNumber` function (Listing 4.21). Those names' order is the same as the occurrence order of agent behaviour code specified in the Alvis model's code layer. In the case of agents sharing the same behaviour in the code layer, their order is as they appear in the list of names after the keyword **`agent`**. The code Listing 4.31 depicts a generic template used by the compiler to generate the type assuming that the modelled system consists of four agents, namely FirstAgent, SecondAgent, ThirdAgent, and ForthAgent. For simplicity and readability reasons, there is no name mangling involved, or no dedicated module produced by default and the type is generated in the Main module together with all other symbols. Therefore, the modeller is responsible for keeping the agents' names distinct from other existing model's or Haskell's keywords or already defined symbols. Additionally, Listing 4.32 shows the `Agent` data type generated for the Producer-Consumer problem.

```
data Agent
  = FirstAgent
  | SecondAgent
  | ThirdAgent
  | FourthAgent
  deriving (Eq,Ord,Show)
```

Listing 4.31: Agent algebraic data type template. The names FirstAgent, SecondAgent, . . . are replaced with the specific agent names.

```
data Agent
  = Producer
  | Buffer
  | Consumer
  deriving (Eq,Ord,Show)
```

Listing 4.32: Agent type example for the Producer-Consumer problem

4.2 Transitions

Transitions are the edges or arcs of the LTS graph. They represent events that can occur in the system that imply changes in the state of the whole system from one to another. In the Haskell representation of the modelled system, the transition has a dedicated type of `TTranstion`, shown in Listing 4.33. The value constructors of the type `TTranstion` cover the set of all possible transitions that can occur in the system. Not always all types of transitions are present for a particular model. All types of transition but one correspond to the execution of a single statement of an Alvis code. The only transition of `STTime` corresponds to no particular statement of the Alvis code but to the mere passage of time and only within the context of a model with time.

```
data TTransition
  = TDelay Agent Int
  | TExec Agent Int
  | TExit Agent Int
  | TIn Port Int
  | TInAP Port Port Int
  | TInPP Port Port Int
  | TInF Port Port Int
  | TJump Agent Int
  | TLoop Agent Int
  | TLoopEvery Agent Int
  | TNull Agent Int
  | TOut Port Int
  | TOutAP Port Port Int
  | TOutPP Port Port Int
  | TOutF Port Port Int
  | TSelect Agent Int
  | TStart Agent Int
  | STInAP Port Port Int
```

```
    | STInPP Port Port Int
    | STOutAP Port Port Int
    | STOutPP Port Port Int
    | STDelayEnd Agent Int
    | STLoopEnd Agent Int
    | STInEnd Port Int
    | STOutEnd Port Int
    | STTime Int
    deriving (Eq,Ord)
```

Listing 4.33: Definition of the data type TTransition

Every transition that can be performed in the system can be unambiguously identified by the name of the agent that executes the statement related to the transition, the value of the program counter that indicates the statement within the agent's block of code, and the port of another agent with whom the communication channel is established. The last piece of information is required only for the transitions related to a communication instruction. The information about the agent is stored directly in the first parameter of the type `Agent`, cf. Sec. 4.1.9, or indirectly in the first parameter of the type `Port` (but can be extracted by the function `agentName`, cf. Sec. 4.22). The information about the value of the program counter is always stored by the last field of type **`Int`**. Some communication transitions require, however, additional information of the remote port with which the communication is established. That information is conveyed by the second parameter of the type `Port`, available only in the following transitions: `TInAP`, `TInPP`, `TInF`, `TOutAP`, `TOutPP`, `TOutF`, `STDelayEnd`, `STLoopEnd`, `STInEnd`, and `STOutEnd`. The only exception to the above description is the transition `STTime`, for which the only argument is of the type **`Int`** and describes the number of time units that passed.

Transition value constructors `TDelay`, `TExec`, `TExit`, `TJump`, and `TStart` correspond to the execution of instructions **`delay`**, **`exec`**, **`exit`**, **`jump`**, **`start`**. Transition value constructor `TLoop` corresponds to testing of the guard expression (if any is present) of **`loop`** instruction and a conditional entrance into the associated code block of instructions or skipping over it. If the condition is not present the loop is unconditional and execution always enters the associated code block. Transition value constructor `TLoopEvery` corresponds to check of the timer status of **`loop every`** and entering into the associated code block of instructions or resuming execution of the agent after time-out of the timer. Transition value constructor `TSelect` corresponds to the evaluation of all the guards in the sequence of their appearance in the Alvis code layer and entrance into the first associated block of statements related to the first open branch or control is transferred to the very next statement after the last guard's block of statements if all guards are closed. All transition value constructors of type `TInAP`, `TInPP`, `TInF`, and `TIn` correspond to various phases of execution of a single **`in`** statement. There are as many as four different types of transition because execution of the statement **`in`** involves communication with another agent in the system. After initialisation of execution of the statement the following different results may occur:

- `TInAP A_p P_q x` – a successful call of the procedure `P_q` of the passive agent `P` is initiated by the active agent `A` and the control is transferred to the first statement of the called procedure, cf. definition E9. The agent `A` executes its x^{th} instruction which is the instruction **`in`** `p`.
- `TInPP P_p Q_q x` – a successful call of the procedure `Q_q` of the passive agent is initiated by another passive agent `P` and the control is transferred to the first statement of the called procedure, cf. definition E10. The agent `P` executes its x^{th} instruction which is the instruction **`in`** `p`.
- `TInF A_p x` – the active agent successfully completes communication with another active agent that had already commenced the communication before and had been waiting for its finalisation (see below, and cf. definition E11). The agent `A` executes its x^{th} instruction which is the instruction **`in`** `p`.
- `TIn A_p x` – there are three different events that are umbrellaed under the name of this transition, namely:
  - the active agent `A` executes the instruction **`in`** `p` but there is no available procedure to be called of any connected passive agent nor other active agent had already initialised communication nor had been waiting for finalisation of the communication (see above and cf. definition E7).
  - the passive agent `A` executes the instruction **`in`** `p` on the non-procedural port `A_p` but there is no available procedure to be called of any connected passive agent (cf. definition E8).
  - the passive agent `A` executes the instruction **`in`** `p` on its procedural port `A_p` and therefore, acquires the value from the agent that called the procedure (cf. definition E8).

All transition value constructors of type `TOutAP`, `TOutPP`, `TOutF`, and `TOut` correspond to various phases of execution of the single statement of **`out`**. The situation is similar to the case of the statement **`in`** considered just before. There are as many as four transitions because execution of the statement **`out`** ↪ involves communication with another agent in the system. After initialisation of the execution of the statement, the following different results may occur:

- `TOutAP A_p P_q x` – a successful call of the procedure `P_q` of the passive agent `P` is initiated by the active agent `A` and the control is transferred to the first statement of the called procedure, cf. definition E22. The agent `A` executes its x^{th} instruction which is the instruction **`out`** `p`.
- `TOutPP P_p Q_q x` – a successful call of the procedure `Q_q` of the passive agent is initiated by another passive agent `P` and the control is transferred to the first statement of the called procedure, cf. definition E23. The agent `P` executes its x^{th} instruction which is the instruction **`out`** `p`.
- `TOutF A_p x` – the active agent successfully completes communication with another active agent that had already commenced the communication before and had been waiting for its finalisation (see below, and cf. definition E24). The agent `A` executes its x^{th} instruction which is the instruction **`out`** ↪ `p`.

- `TOut A_p x` – there are three different events that are umbrellaed under the name of this transition, namely:
  - the active agent `A` executes the instruction **`out`** `p` but there is no available procedure to be called of any connected passive agent nor other active agent had already initialised communication nor had been waiting for finalisation of the communication (see above and cf. definition E20).
  - the passive agent `A` executes the instruction **`out`** `p` on the non-procedural port `A_p` but there is no available procedure to be called of any connected passive agent (cf. definition E21).
  - the passive agent `A` executes the instruction **`out`** `p` on its procedural port `A_p` and therefore, acquires the value from the agent that called the procedure (cf. definition E21).

Additionally to the previously discussed transitions, for the back compatibility reason and simplification of description, there were introduced four transitions with value constructors of `STInAP A_p Q_q x`, `STInPP A_p Q_q x`, `STOutAP A_p Q_q x`, and `STOutPP A_p Q_q x`. They correspond to the situation when the procedure `Q_q` had been taken before the execution of the **`in`** `p` or **`out`** `p`. Therefore, the system executed transitions `TIn A_p x` or `TOut A_p x` when procedure `Q_q` was not yet available and agent `A` is waiting. Now, the procedure `Q_q` becomes available again, and there is a need to wake up the active or passive agent `A` that is waiting on the procedure `Q_q` to be available.

The statements **`delay`**, last **`null`** instruction of the associated **`loop every`** block, non-blocking **`in`** ↪ , and non-blocking **`out`** may possibly move the agent that executes them into the waiting mode with a timer counting down to the moment of the unconditional resumption of the agent execution. The timer (the value constructor of `CTimer` of the type `ContextInfo`) is inserted into agent's context list (Sec. 4.1.5). After time-out, the value constructor `CTimeout`, cf. Sec. 4.1.5 replaces the timer in the context list. The presence of `CTimeout` entry, eventually, enables the following transitions: `STDelayEnd`, `STLoopEnd`, `STInEnd`, `STOutEnd`.

The last transition of `STTime` corresponds to the sheer passage of time with no other action simultaneously happening in the system. This condition may occur when, e.g. all agents are in the waiting mode, and there is no action to be performed but after few time units at least one agent is to be woken up (that is why it is different from deadlock situation).

All the value constructors of the type of `TTransition` are equipped with a custom conversion to its string representation. The conversion method follows Haskell standard approach of instantiation of the **`Show`** type class as may be verified in Listing 4.34.

```haskell
instance Show TTransition where
  show (TDelay a _) = "delay(" ++ show a ++ ")"
  show (TExec a _) = "exec(" ++ show a ++ ")"
  show (TExit a _) = "exit(" ++ show a ++ ")"
  show (TIn p _) = "in(" ++ show p ++ ")"
  show (TInAP p _ _) = "in(" ++ show p ++ ")"
show (TInPP p _ _) = "in(" ++ show p ++ ")"
show (TInF p _ _) = "in(" ++ show p ++ ")"
show (TJump a _) = "jump(" ++ show a ++ ")"
show (TLoop a _) = "loop(" ++ show a ++ ")"
show (TLoopEvery a _) = "loop_every(" ++ show a ++ ")"
show (TNull a _) = "null(" ++ show a ++ ")"
show (TOut p _) = "out(" ++ show p ++ ")"
show (TOutAP p _ _) = "out(" ++ show p ++ ")"
show (TOutPP p _ _) = "out(" ++ show p ++ ")"
show (TOutF p _ _) = "out(" ++ show p ++ ")"
show (TSelect a _) = "select(" ++ show a ++ ")"
show (TStart a _) = "start(" ++ show a ++ ")"
show (STInAP p _ _) = "wakeup(" ++ show p ++ ")"
show (STInPP p _ _) = "wakeup(" ++ show p ++ ")"
show (STOutAP p _ _) = "wakeup(" ++ show p ++ ")"
show (STOutPP p _ _) = "wakeup(" ++ show p ++ ")"
show (STDelayEnd a _) = "timeout(" ++ show a ++ ")"
show (STLoopEnd a _) = "timeout(" ++ show a ++ ")"
show (STInEnd p _) = "timeout(" ++ show p ++ ")"
show (STOutEnd p _) = "timeout(" ++ show p ++ ")"
show (STTime _) = "time"
```

Listing 4.34: ***Show*** instance of TTransition ***data*** type

4.3 Function enable

The function `enable` is one of the two functions that implicitly represent the whole LTS state graph of the modelled system. The function `enable` calculates what kind of transitions are available in the given system state and for the given agent. In terms of the Haskell model, the function `enable` returns the list of values of the type `TTransition`, cf. Sec. 4.2. The function `enable` requires a value of the type `State`, cf. Sec. 4.1.1, and a value of the type `Agent`, cf. Sec. 4.1.9. The function `enable` has the type signature, as presented in Listing 4.1.

The body of the function `enable` is a long switch-case like specification. Therefore, on the left-hand side of the single equal sign (assignment operator in Haskell), there are all conditions required to fully restrict all possible states of the system to the subset of states for which proper transitions are enabled. On the right-hand side of the equal sign, there are transitions to be returned when all the conditions are met. Generated conditions follow the E rules from E1 up to E39. The example of the function `enable` generated by the Alvis compiler for the Producer-Consumer problem is presented in Listing 4.35.

```
enable :: State -> Agent -> [TTransition]

enable s@(state1@(am1,pc1,ci1,pv1@(pv1_1)),state2@(am2,pc2,ci2,pv2@(pv2_1, pv2_2)),state3@(am3,pc3,ci3,pv3@(pv3_1))) Producer
  | am1 == X && pc1 == 1 = [TLoop Producer 1]
  | am1 == X && pc1 == 2 = [TExec Producer 2]
  | am1 == X && pc1 == 3 && (procFree ci1) && am2 == W && member (CIn Buffer_put) ci2 = [TOutAP Producer_push Buffer_put 3]
  | am1 == X && pc1 == 3 && (procFree ci1) = [TOut Producer_push 3]
  | am1 == W && pc1 == 3 && member (COut Producer_push) ci1 && am2 == W && member (CIn Buffer_put) ci2 = [STOutAP
        Producer_push Buffer_put 3]
  | otherwise = []

enable s@(state1@(am1,pc1,ci1,pv1@(pv1_1)),state2@(am2,pc2,ci2,pv2@(pv2_1, pv2_2)),state3@(am3,pc3,ci3,pv3@(pv3_1))) Buffer
  | am2 == T && pc2 == 1 && (xContext 2 s) = [TOut Buffer_get 1]
  | am2 == T && pc2 == 2 && (xContext 2 s) = [TExec Buffer 2]
  | am2 == T && pc2 == 3 && (xContext 2 s) = [TExit Buffer 3]
  | am2 == T && pc2 == 4 && (xContext 2 s) = [TIn Buffer_put 4]
  | am2 == T && pc2 == 5 && (xContext 2 s) = [TExec Buffer 5]
  | am2 == T && pc2 == 6 && (xContext 2 s) = [TExit Buffer 6]
  | otherwise = []
```

Listing 4.35: Function enable generated for the Producer-Consumer problem (For brevity, Consumer part omitted)

The Alvis compiler generates the definition of the function `enable` by the algorithm presented in Algorithm 4.1. The algorithm takes advantage extensively of the so-called string interpolation. String interpolation is a substitution mechanism that replaces every occurrence of the `{variable}` within the string literal with the string representation of the `variable`'s value, e.g., string literal is as follows: `"abc:␣{abc}"`, the value of abc is 7, then after interpolation the value of the string will be `"abc:␣7"`.

The algorithm starts the code generation from printing the function `enable` type signature, cf. line 1. The central part of the algorithm is the loop that explores all instructions of every agent in the system. For every agent the definition of the case of the function `enable` is printed with a state of the system decomposed by pattern matching and agent limited by the correct value of the type `Agent`, cf. line 4. Later on, if-then cases are generated for every instruction of that agent by the internal loop. The process of cases generation is realized by one of the three sub-algorithms presented in Algorithms 4.2 to 4.4.

The last step is to print the default case when the state of the system does not match any condition where the transition should be available, and therefore, there are no transitions enabled, cf. line 19. The default condition can be reached when the mode of an active agent is **I**nit.

```
1  print "enable :: State –> Agent –> [TTransition]"
2  for each agent in modelled system then
3    systemStatePatternMatching ← string of pattern match to decompose the system state
4    print "enable {systemStatePatternMatching} {agentName}"
5    index ← index of agent
6    for each possible pcValue in agent_code_block then
7      alvis_instruction ← instruction at pc in agent_code_block
8      transition ← value constructor of the type TTransition related to the alvis_instruction
9      if alvis_instruction is not a communication instruction then
10         call enable non–communication instruction, cf. Algorithm 4.2
```

```
11      else
12          if the agent is active then
13              call enable communication instruction of active agent, cf. Algorithm 4.3
14          else
15              call enable communication instruction of passive agent, cf. Algorithm 4.4
16          end
17      end
18    end
19    print " | otherwise = []"
20  end
```

Algorithm 4.1: Algorithm generating the function enable.

The first algorithm solves the problem of generation cases for the non-communication instructions, namely: **`delay`**, **`exec`**, **`exit`**, **`loop`**, **`null`**, **`select`**, and **`start`**. The algorithm is presented in Algorithm 4.2. According to the common assumptions for **E** rules for active agents, Sec. 3.5, for every non-communication instruction, the algorithm generates the following conditions: the agent is in the running mode, and the agent's program counter indicates the instruction to be executed. Moreover, for every such instruction, the algorithm generates the only reachable transition corresponding to that instruction, cf. line 2. According to the rules E1 and E16, for the instructions **`delay`**, and **`loop every`**, there are additionally generated system transitions restricted to the state when the agent is in the waiting mode, the agent's program counter indicates the mentioned instruction and the time-out event has been already generated, cf. lines 4 up to 10.

According to the common assumptions for **E** rules for passive agents for every non-communication instruction, the algorithm generates the condition that passive agent is in the taken mode, the agent's program counter indicates the instruction to be executed, and the context agent's mode is running. Moreover, the algorithm additionally generates the only reachable transition, cf. line 12. According to the rules E2 and E17, for the instructions **`delay`**, and **`loop every`**, there are additionally generated system transitions restricted to the state when the agent is in the taken mode, the agent's program counter indicates the mentioned instruction, the context agent's mode is waiting, and the time-out event has been already generated, cf. lines 14 up to 20.

```
1  if the agent is active then
2      print " | am{index} == X && pc{index} == {pcValue} = [{transition} {agentName} {pcValue}]"
3      * print additional system transitions in the case of delay and loop (every ...) instructions
4      if alvisInstruction is the instruction delay then
5          systemTransition ← STDelayEnd
6          print " | am{index} == W && pc{index} == {pcValue} && member (CTimeout {pcValue}) ci{index} = [{systemTransition} {
                ↪ agentName} {pcValue}]"
7      else if alvisInstruction is the instruction loop every then
8          systemTransition ← STLoopEnd
9          print " | am{index} == W && pc{index} == {pcValue} && member (CTimeout {pcValue}) ci{index} = [{systemTransition} {
                ↪ agentName} {pcValue}]"
```

```
        end
    else
        print " | am{index} == T && pc{index} == {pcValue} && xContext {index} s = [{transition} {agentName} {pcValue}]"
        * print additional system transitions in the case of delay or loop (every ...) instructions
        if alvisInstruction is the instruction delay then
            systemTransition ← STDelayEnd
            print " | am{index} == T && pc{index} == {pcValue} && wContext {index} s && member (CTimeout {pcValue}) ci{index} = [{
              ↪ systemTransition} {agentName} {pcValue}]"
        else if alvisInstruction is the instruction loop every then
            systemTransition ← STLoopEnd
            print " | am{index} == T && pc{index} == {pcValue} && wContext {index} s && member (CTimeout {pcValue}) ci{index} = [{
              ↪ systemTransition} {agentName} {pcValue}]"
        end
    end
```

Algorithm 4.2: Algorithm generating the function enable. Agent enabling non-communication transitions.

The second and third algorithm solve the problem of the generation part of the enable function related to the communication instructions, namely: **`in`** and **`out`** in both variants of blocking and non-blocking mode. The algorithm that is specialized for the active agent is presented in Algorithm 4.3 and the variant for the passive agent in Algorithm 4.4.

According to the definitions for active agents E7, E9, and E11, for every communication instruction, algorithm generates the common condition that agent is in the running mode, and the agent's program counter indicates the instruction being executed, and no procedure is already being executed, cf. line 37. Moreover, for every connected procedure and every connected active agent, the condition and related transition are generated and collected into two lists: `extraConditions` and `resultingTransitions` respectively, cf. lines 5 up to 35. The generated condition is the same for connected active agent's port and passive agent's procedure, cf. line 15. That is because when a remote agent is passive, then its procedure has to be available. Therefore, the passive agent has to be in the waiting mode, and the entry related to the procedure has to be a member of its context list. Likewise, when the remote agent is active, it had had to initialize the communication before. Therefore, the remote agent has to be in the waiting mode, and the entry signifying the suspended communication has to be present in its context list. Additionally, for a remote active agent, it has to be excluded the possibility that the non-blocking **`in`** or **`out`** instruction is timed-out, cf. line 17. The generated transition represents the type of the remote agent and the type of the local instruction (input or output), cf. line 33. From the generated sets of `extraConditions` and `resultingTransitions` there are generated all possible k-combinations for all values of k but 0, starting from $card(extraConditions)$ down to 1, cf. line 39. Later on, the combination of conditions is concatenated into a single condition with the logical operator representing conjunction, cf. line 40, and the respective combination of transitions are concatenated into a list of enabled transitions in that state of the system, cf. line 41. Finally, the combination of conditions together with the common condition, and list of enabled transitions are printed as a case of the enable function, cf. line 42.

The previously described algorithm for the generation of cases dealing with all possible combination of connections between two agents is similar to the algorithm that generates cases for the system transitions, cf. lines 53 up to 89. This part of the algorithm is intended to cover the rules E29 and E31. For every communication instruction, algorithm generates the common condition that agent is in the waiting mode, and the agent's program counter indicates the instruction being suspended, and no procedure is already being executed, cf. line 83. The system transitions are enabled in order to wake up the active agent that is in the waiting mode because the passive agent's procedure becomes now available. Therefore, in this case, the algorithm loops only over connected procedures and generates condition and transition for every callable procedure. Then, like before, all possible combination of conditions are explored and the respective combination of transitions. Finally, the combination of extra conditions together with common condition, and all enabled transitions are printed as a case of the enable function, cf. line 88.

```
1  extraConditions ← []
2  resultingTransitions ← []
3  resultingSystemTransitions ← []
4
5  for each callable procedure or callable active agent's port then
6    remotePort ← name of called procedural port or called active agent's port of type Port
7    remoteIndex ← index of the called agent
8
9    if alvisInstruction is input one then
10     contextInfoCtor ← COut
11   else
12     contextInfoCtor ← CIn
13   end
14
15   newCondition ← "am{remoteIndex} == W && member ({contextInfoCtor} {remotePort}) ci{remoteIndex}"
16   if called agent is active then
17     newCondition ← "{newCondition} && notMember (CTimeout {pc}) ci{remoteIndex}"
18   end
19   extraConditions ← extraConditions ⊕ newCondition
20
21   if alvisInstruction is input and remote agent is passive then
22     transition ← TInAP
23   else if alvisInstruction is input and remote agent is active then
24     transition ← TInF
25   else if alvisInstruction is output and remote agent is passive then
26     transition ← TOutAP
27   else
28     transition ← TOutF
29   end
30
31   port ← port name as the value of the type Port from communication instruction
32
```

```
    newTransition ← "{transition} {port} {remotePort} {pcValue}"
    resultingTransitions ← resultingTransitions ⊕ newTransition
end

commonCondition ← "am{index} == X && pc{index} == {pcValue} && procFree ci{index}"

for each conditionCombination ← $2^{extraConditions}/\varnothing$ and transitionCombination ← $2^{resultingTransitions}/\varnothing$ both ordered
        ↪ descendingly by cardinality of the subsets then
    conditions ← concatenate conditionCombination with "&&" as separator
    transitions ← concatenate transitionCombination with "," as separator
    print " | {commonCondition} && {conditions} = [{transitions}]"
end

* finally print enable case when active agents move into waiting mode due to unavailable communication channels
if alvisInstruction is input then
    transition ← "TIn"
else
    transition ← "TOut"
end
print " | {condition} = [{transition} {port} {pcValue}]"

for each callable procedure only then
    remotePort ← called procedural port
    remoteIndex ← index of the called passive agent

    if alvisInstruction is input one then
        contextInfoCtor ← COut
    else
        contextInfoCtor ← CIn
    end

    newCondition ← "am{remoteIndex} == W && member ({contextInfoCtor} {remotePort}) ci{remoteIndex}"
    extraConditions ← extraConditions ⊕ newCondition

    if alvisInstruction is input then
        systemTransition ← STInAP
    else
        systemTransition ← STOutAP
    end

    port ← port name of type Port from communication instruction

    newSystemTransition ← "{systemTransition} {port} {remotePort} {pcValue}"
    resultingSystemTransitions ← resultingSystemTransitions ⊕ newSystemTransition
end

if alvisInstruction is input one then
    contextInfoCtor ← CIn
else
```

```
    contextInfoCtor ← COut
*end*
commonCondition ← "am{index} == W && pc{index} == {pcValue} && member ({contextInfoCtor} {port}) ci{index}"

*for each* conditionCombination ← $2^{extraConditions}/\varnothing$ *and* transitionCombination ← $2^{systemTransitions}/\varnothing$ both ordered descendingly
        ↪ by cardinality of the subsets *then*
    conditions ← *concatenate* condition_combination *with* "&&" *separator*
    transitions ← *concatenate* transition_combination *with* "," *separator*
    *print* " | {commonCondition} && procFree ci{index} && {conditions} = [{transitions}]"
*end*
```

Algorithm 4.3: Algorithm generating the function enable. Active agent enabling communication transitions

```
extraConditions ← []
resultingTransitions ← []
resultingSystemTransitions ← []

*for each* callable procedure only *then*
    remotePort ← called the procedural port name of type Port
    remoteIndex ← index of the called passive agent

    *if* alvisInstruction is input one *then*
        contextInfoCtor ← COut
    *else*
        contextInfoCtor ← CIn
    *end*

    newCondition ← "am{remoteIndex} == W && member ({contextInfoCtor} {remotePort}) ci{remoteIndex}"
    extraConditions ← extraConditions ⊕ newCondition

    *if* alvisInstruction is input *and* called agent is passive *then*
        transition ← "TInPP"
    *else if* alvisInstruction is output *and* called agent is passive *then*
        transition ← "TOutPP"
    *end*

    port ← port name of type Port *from* communication instruction

    * e.g. TInPP AgentName_port RemoteAgent_port 17
    newTransition ← "{transition} {port} {remotePort} {pcValue}"
    resultingTransitions ← resultingTransitions ⊕ newTransition
*end*


condition ← "am{index} == T && pc{index} == {pcValue}"

*for each* conditionCombination ← $2^{extraConditions}/\varnothing$ *and* transitionCombination ← $2^{systemTransitions}/\varnothing$ both ordered descendingly
        ↪ by cardinality of the subsets *then*
    conditions ← *concatenate* conditionCombination *with* "&&" as *separator*
```

```
  transitions ← concatenate transitionCombination with "," as separator
  print " | {condition} && {conditions} = [{transitions}]"
end

* finally print enable case when the passive agent moves its context active agent into waiting mode due to unavailable communication
    ↪ channels
if alvisInstruction is input then
  transition ← TIn
else
  transition ← TOut
end

print " | {condition} && (xContext {index} s) && (procFree ci{index}) = [{transition} {port} {pcValue}]"

for each callable procedure only
  remotePort ← called procedural port
  remoteIndex ← index of the called passive agent

  if alvisInstruction is input one then
    contextInfoCtor ← COut
  else
    contextInfoCtor ← CIn
  end

  newCondition ← "am{remoteIndex} == W && member ({contextInfoCtor} {remotePort}) ci{remoteIndex}"
  extraConditions ← extraConditions ⊕ newCondition

  if alvisInstruction is input then
    systemTransition ← STInAP
  else
    systemTransition ← STOutAP
  end

  port ← port name of type Port from communication instruction

  newSystemTransition ← "{systemTransition} {port} {remotePort} {pcValue}"
  resultingSystemTransitions ← resultingSystemTransitions ⊕ newSystemTransition
end

if alvisInstruction is input one then
  contextInfoCtor ← CIn
else
  contextInfoCtor ← COut
end
condition ← "am{index} == T && pc{index} == {pcValue} && (wContext 1 s) && member ({contextInfoCtor} {port}) ci{index}"

for each conditionCombination ← $2^{extraConditions}/\varnothing$ and transitionCombination ← $2^{systemTransitions}/\varnothing$ both ordered descendingly
    ↪ by cardinality of the subsets then
  conditions ← concatenate conditionCombination with "&&" as separator
  transitions ← *concatenate* transitionCombination *with* "," as *separator*
  *print* " | {condition} && procFree ci{index} && {conditions} = [{transitions}]"
end
```

Algorithm 4.4: Algorithm generating the function enable. Passive agent enabling communication transitions.

4.4 Function fire

The function `fire` is the second function of the two functions that implicitly represent the whole LTS state graph of the modelled system. The function `fire` calculates the new system states reachable from the given system state by the given transition. The transition is expected to be enabled in that state. Otherwise, the behaviour is undefined. In terms of the Haskell model, the function `fire` returns a list of values of the type `State`, cf. Sec. 4.1.1. The function `fire` requires a value of the type `TTransition`, cf. Sec. 4.2 and a value of the type `State`. As a reminder, the function `fire` has the following type definition as already stated in Sec. 4.2.

```
fire :: TTransition -> State -> [State]
```

Listing 4.36: Type definition of the function fire

In most cases, the resulting list is a singleton list, i.e., it contains precisely one system state, but there is the case involving `pick` function, where there could be more than one state fired. Therefore, the system changes its state deterministically for the given transition and the indeterminism can be introduced by the function `pick`.

The body of the function `fire` is a long if-then like specification where on the left-hand side of the single equal sign, there are all conditions required to restrict all possible states of the system to the subset of the desired system state. On the right-hand side of the equal sign, there is a list of updated system states to be returned when all the conditions are met, most of the time a singleton list. In the definition of the function `fire`, extensive usage of the Haskell pattern matching can be observed. Most of the conditions are represented by the pattern matching only and are of the following type: if the transition is of the desired value (i.e., the type of the executed instruction, the agent that executes it, and the value of the agent's program counter is as required, e.g., `TLoop Producer 1`), and the system state is given then the updated system state is as stated. For some transitions, it is required to provide additional predicates, further constraining the system state. That is achieved by providing an additional predicate based on the value of the system state before-the-modification. The example of the function `fire` body is presented in Listing 4.37. The example is provided for the Producer-Consumer problem, presented in Sec. 1.9.

In the following sections, the algorithms required by the main loop presented in Algorithm 4.5 are discussed. The algorithms are written in a pseudocode and frequently take advantage of the following subprocedures that simplify them:

- *systemStatePatternMatching* – is a function argument that allows decomposition of a variable representing system state into its constituents. Additionally, it leverages a Haskell syntax that allows addressing a matched variable as a whole and to deconstruct it by some relevant value constructor at the same time, here: a tuple constructor. The variable is addressed by a name on the left-hand side of `@` operator, and variable decomposition into their elements is on the right-hand side of that operator. An example of a current state of a hypothetical model with four agents could be:

  ```
  s@(state1@(am1,pc1,ci1,pv1@(pv1_1,pv1_2)), state2@(am2,pc2,ci2,pv2@
     ↪ ()), state3@(am3,pc3,ci3,pv3@(pv3_1,pv3_2,pv3_3)), state4@(am4
     ↪ ,pc4,ci4,pv4@()))
  ```

  Here, `s` represents the whole system state, while on the right-hand side is a four tuple decomposition of states of particular agents. Elements of that tuple are further decomposed, i.e. a state of the agent is decomposed into its mode, program counter, context list, and its variables. The tuple of agent variable is further decomposed and allows addressing particular variable by synthetic name `pvX_Y`.
- *newSystemState* ***with*** *text* ***at*** *index* – is a pseudocode of generator that creates a tuple that represents a new system state by copying variables from *systemStatePatternMatching*. Let us consider an example *newSystemState* ***with*** *XYZ* ***at*** *3* will generate a tuple that copies all agents state without modification but replacing agent at index with a given text:

  ```
  (state1, state2, XYZ, state4)
  ```

- *expression rewrite* – is a pseudocode that allows creating valid Haskell expression from Alvis general expression by replacing agent local variable names with synthetic variable names from *systemStatePatternMatching*. The expression is copied as is, but all names of the local variables are translated. The translation rule is as follows: all variables have a new name: $pv\{agentIndex\}\_\{varIndex\}$, where $agentIndex$ is the agent index of the agent that executes the instruction and $varIndex$ is a sequence number as indicated by order of definition of the variables in the agent. E.g., if the compiler encounters the following instruction: **`exec`** `y = abs(a + b) *` `↪ c + 7;`, and variables a, b, and y are the local agent variables, while symbols c, and abs are from Haskell modules. Then, it rewrites the expression into

  ```
  abs(pv3_1 + pv3_2) * c + 7
  ```

  This is a non-trivial task and parsing such expressions required to implement the whole Haskell syntax parser in the Alvis compiler which was done by the author and an expression parser is present in the Alvis compiler.

- *newVariables* ***of*** *index* ***with*** *expression* ***at*** *varIndex* – is a pseudocode generator that creates a tuple of all variables that belong to agent with index. If an expression starting with ***with*** is omitted, the following tuple is generated: `(pv{`**`index`**`}_1,pv{`**`index`**`}_2,...)`. If that expression is present, then the variable `pv{`**`index`**`}_{varIndex}` is replaced with *expression* (assuming $varIndex = 2$):

```
(pv{index}_1,abs(pv3_1 + pv3_2) * c + 7,pv{index}_3, ...)
```

```
fire :: TTransition -> State -> [State]

fire (TLoop Producer 1) s@(state1@(am1,pc1,ci1,pv1@(pv1_1)),state2@(am2,pc2,ci2,pv2@(pv2_1, pv2_2)),state3@(am3,pc3,ci3,pv3@(
    ↪ pv3_1)))
  = [((am1,2,ci1,pv1),state2,state3)]

fire (TExec Producer 2) s@(state1@(am1,pc1,ci1,pv1@(pv1_1)),state2@(am2,pc2,ci2,pv2@(pv2_1, pv2_2)),state3@(am3,pc3,ci3,pv3@(
    ↪ pv3_1)))
  = [((am1,3,ci1,(pvX_1)),state2,state3) | pvX_1 <- [1..3]]

fire (TOutAP Producer_push Buffer_put 3) s@(state1@(am1,pc1,ci1,pv1@(pv1_1)),state2@(am2,pc2,ci2,pv2@(pv2_1, pv2_2)),state3@(
    ↪ am3,pc3,ci3,pv3@(pv3_1)))
  = [((am1,pc1,insert (CProc Buffer_put) ci1,pv1),(T,4,empty,pv2),state3)]

fire (TOut Producer_push 3) s@(state1@(am1,pc1,ci1,pv1@(pv1_1)),state2@(am2,pc2,ci2,pv2@(pv2_1, pv2_2)),state3@(am3,pc3,ci3,
    ↪ pv3@(pv3_1)))
  = [((W,pc1,insert (COut Producer_push) ci1,pv1),state2,state3)]

fire (STOutAP Producer_push Buffer_put 3) s@(state1@(am1,pc1,ci1,pv1@(pv1_1)),state2@(am2,pc2,ci2,pv2@(pv2_1, pv2_2)),state3@(
    ↪ am3,pc3,ci3,pv3@(pv3_1)))
  | isAwakableProducer state1 Buffer_put = [(awakeProducer state1 Buffer_put,(T,4,empty,pv2),state3)]
```

Listing 4.37: Function fire as generated for the Producer-Consumer problem (for clarity, agent Buffer's and agent Consumer's parts are omitted)

The Alvis compiler generates the body of the function `fire` by the algorithm presented in Algorithm 4.5. The code is generated when the **`print`** method is called with an argument of type string. Then the string is printed to the output file with the **IHR** model. The generated function is model specific because of the hard-coded values and model-specific information like values of program counters, which instruction follows executed one, or the structure of communication channels to name a few. The model agnostic function `fire` would be far more complicated and far harder to be adjusted by engineers after its generation by the compiler because all the logic of code generation that now is inside the Alvis compiler would have to be moved to the resulting **IHR** model.

```
print "fire :: TTransition -> State -> [State]"

for each transition as generated by the function enable then
    nextPc ← pc of next instruction to execute after instruction is successful (if there is not any then LAST)
```

```
index ← index of agent executing this instruction
systemStatePatternMatching ← *string* of pattern match to decompose the system state
*print* "fire {transition} {systemStatePatternMatching}"
*if* transition is TDelay *then*
  ∗ *see* 4.6
  call generate_delay_trasition
*else if* transition is TExec *then*
  ∗ *see* 4.7
  call generate_exec_transition
*else if* transition is TExit *then*
  ∗ *see* 4.8
  call generate_exit_transition
*else if* transition is TIn *then*
  ∗ *see* 4.9
  call generate_in_transition
*else if* transition is TInAP *then*
  ∗ *see* 4.10
  call generate_in_active_passive_transition
*else if* transition is TInPP *then*
  ∗ *see* 4.11
  call generate_in_passive_passive_transition
*else if* transition is TInF *then*
  ∗ *see* 4.12
  call generate_in_finilise_transition
*else if* transition is TJump *then*
  ∗ *see* 4.13
  call generate_jump_transition
*else if* transition is TLoop *then*
  ∗ *see* 4.14
  call generate_loop_transition
*else if* transition is TLoopEvery *then*
  ∗ *see* 4.15
  call generate_loop_every_transition
*else if* transition is TNull *then*
  ∗ *see* 4.16
  call generate_null_transition
*else if* transition is TOut *then*
  ∗ *see* 4.17
  call generate_out_transition
*else if* transition is TOutAP *then*
  ∗ *see* 4.18
  call generate_out_active_passive_transition
*else if* transition is TOutPP *then*
  ∗ *see* 4.19
  call generate_out_passive_passive_transition
*else if* transition is TOutF *then*
  ∗ *see* 4.20
  call generate_out_finalisation_transition
*else if* transition is TSelect *then*
```

```
        * see 4.21
        call generate_select_transition
    else if transition is TStart then
        * see 4.22
        call generate_start_transition
    else if transition is STInAP or STOutAP then
        * see 4.23
        call generate_in_active_passive_system_transition
    else if transition is STInPP then
        * see 4.24
        call generate_in_passive_passive_system_transition
    else if transition is STOutPP then
        * see 4.26
        call generate_out_passive_passive_system_transition
    else if transition is STDelayEnd then
        * see 4.27
        call generate_delay_end_system_transition
    else if transition is STInEnd then
        * see 4.28
        call generate_in_end_system_transition
    else if transition is STLoopEnd then
        * see 4.29
        call generate_loop_end_system_transition
    else if transition is STOutEnd then
        * see 4.30
        call generate_out_end_system_transition
    end
end
```

Algorithm 4.5: Function fire generation algorithm

4.4.1 Transition TDelay

The result of execution of the transition **TDelay** on the agent's state depends on the type of the agent. There is a difference because the instruction suspends execution of the agent that executes the code directly on the processing unit. Therefore, for an active agent, it is that agent itself, but for a passive agent, this is the context active agent that executes code of the procedure directly or indirectly.

The algorithm is presented in Algorithm 4.6. In terms of the agent's state, cf. line 4, execution of the instruction **`delay`** by an active agent means that the agent's mode is changed to **W**aiting, its program counter is left unmodified, the context item of `CTimer` is inserted into its context list, and the values of the agent's variables are left unmodified.

In the case of the passive agent executing the instruction **`delay`**, cf. line 8, the agent's mode is left unmodified, its program counter is left unmodified, a context item of `CTimer` is inserted into its context list, and the values of agent's variables are left unmodified. Additionally, for every possible context active agent executing the code of the procedure, the state of that agent is conditionally modified too. For each such a case there is generated a dedicated condition assuring that the system is in the state that the to-be-modified agent's state is executing the procedure, cf. line 12. Then, cf. line 11, the mode of the context agent is changed to **W**aiting, the context agent's program counter is left unmodified, the context agent's context list is left unmodified, and values of its variables are left unmodified.

```
1  delay ← delay−time requested – argument of delay instruction
2
3  if the agent is active then
4    newState ← "(W,pc{index},insert (CTimer {pc} {delay}) ci{index},pv{index})"
5    newSystemState ← newSystemState with newState at index
6    print " = {newSystemState}"
7  else
8    newState ← "(am{index},pc{index},insert (CTimer {pc} {delay}) ci{index},pv{index})"
9    procedure ← currently executed procedure
10   for each context agent that is able to execute procedure then
11     newContextAgentState ← "(W,pc{index},ci{index},pv{index})"
12     condition = "isWithinContextOf_{contextAgentName} {procedureName} s"
13     newSystemState ← newSystemState with newState at index and with newContextAgentState at contextAgentIndex
14     print " | {condition} = {newSystemState}"
15   end
16 end
```

Algorithm 4.6: Function fire generation algorithm – transition TDelay

4.4.2 Transition TExec

The result of execution of the transition **TExec** on the agent's state depends on the type of the agent. The only difference is that only active agents can finish its execution after execution of the instruction **`exec`**.

When the transition **TExec A x** is executed by the system, it corresponds to the execution of instruction **`exec`** `y = expression;` at the program counter value of x by the agent A, cf. Sec. 3.2. As the result, the expression on the right-hand side of the assignment operator is evaluated and its result is assigned as a new value to the variable on the left-hand side of the assignment operator, `y` in the example. The expression may contain an arbitrary number of variables that belong to the agent executing the instruction, and supports a subset of Haskell expression syntax. Nonetheless, arbitrary Haskell code execution is supported as a call to the user-defined function. Also, user-defined functions can be attached as a Haskell stand-alone module where the whole Haskell syntax is supported.

There is one particular function of `pick`, that is not a standard Haskell function. The function takes a single value out of the given list of values by random. The algorithm is presented in Algorithm 4.7. In the case of an active agent, if the instruction **`exec`** is the last one, then it finishes its execution. In terms of the agent's state, cf. lines 6 up to 10, it means that the agent's mode remains unchanged or is changed to

Finished in the case of **exec** being the last instruction, its program counter is set to the value of the program counter of the next instruction (*nextPc*), or it is set to 0 in case of the last instruction. Its context list is left unmodified, or context list is cleared in the case of last instruction. When the executed expression is other than `pick`, the tuple of values of the agent is rewritten by *expression rewrite*. For example, let us consider the model of just one active agent A with 3 variables $var1$, $var2$, $var3$ and two types of instructions **exec** at program counter value of 1 and 2, cf. Listing 4.38.

```
x = 7 -- Haskell constant
agent A {
  var1 :: Int = 1;          --varIndex=1
  var2 :: Int = 2;          --varIndex=2
  var3 :: Int = 0;          --varIndex=3

  exec var3 = x + var1 + var2 + 3; --pc = 1
  exec var1 = pick [8,10,var3];    --pc = 2
}
```

Listing 4.38: Example of the model of a single active agent with three variables var1, var2, var3 and a single exec instruction at program counter value of 1

The tuple of all variables before the modification is `(pv1_1,pv1_2,pv1_3)`. The expression is rewritten according to the rewrite rules to: `(pv1_1,pv1_2,x + pv1_1 + pv1_2 + 3)`.

When the expression contains the function `pick`, the list of states is returned (rather than a single state). The resulting list has the same number of elements as the list that is the argument of the function `pick`. Every state is the same, but the variable at the left-hand side of an assignment operator is updated to a single value from the list that was the argument of the function `pick`. Therefore, every resulting state is linked to a single value from the argument list. The second instruction **exec** from the example generates the following Haskell expression that describes a list of system states: `[(F,0,empty,(pv2_1,pv2_2))| pv2_2 ↪ <- [8,10,pv1_3]]`.

```
expression ← expression – argument of exec instruction
parsedExpression ← expression rewrite
varIndex ← index of variable assigned in exec instruction
if expression does not contain pick function then
    newVars ← newVariables of index with parsedExpression at varIndex
    if nextPc is LAST then
        newState ← "(F, 0, empty, {newVars})"
    else
        newState ← "(am{index}, {nextPc}, ci{index}, {newVars})"
    end
    newSystemState ← newSystemState with newState at index
    print " = {newSystemState}"
else
    comprehensionExpression ← "| pv{index}_{varIndex} <– {parsedExpression}"
    newVars ← newVariables
    newState ← "(am{index},{newPc},ci{index},{newVars})"

    newSystemState ← newSystemState with newState at index
    print " = [{newSystemState} {comprehensionExpression}]"
end
```

Algorithm 4.7: Function fire generation algorithm – transition TExec

4.4.3 Transition TExit

The result of the execution of the transition **TExit** on the agent's state depends on the type of the agent. The transition **TExit A x** signifies that the agent A executes the instruction **`exit`** indicated by the program counter value x. The algorithm is presented in Algorithm 4.8.

When the system executes the transition **TExit** that is related to the active agent, then the agent finishes its execution and cannot be restarted later on. In terms of the agent's state, cf. line 2, it means that the agent's mode changes to **F**inished, its program counter is set to 0, its context list is cleared, and the values of the agent's variables are left unmodified.

When the system executes the transition **TExit**, that is related to the passive agent, the procedure that is being executed is completed, and therefore, the passive agent becomes available to be retaken. Additionally, the agent that called the procedure changes its state too, the agent that directly called the procedure is called a *caller-agent*, from now on. If the caller-agent is active and its calling instruction is its last instruction, then the caller-agent finishes its execution. Since the last instruction of a passive agent's procedure has to be the instruction **`exit`**, it is evident that the passive caller-agent has to continue its execution if another passive agent called its procedure, and only the active caller-agent can finish its execution. For both passive and active agents that continue their execution, the information of calling that procedure is removed from their context list, and their program counter is advanced to the next instruction. In terms of the agent's state, cf. line 6, it means that the passive agent's mode is changed to **W**aiting, its program counter is set to 0, its context list is populated with all available procedures based on the current valuation of the agent's variables, and its variables are left unmodified. To simplify the algorithm of the resetting the state of the passive agent back to waiting mode, the generated code delegates the resetting task to the helper method **`init`**, cf. Sec. 4.5.6 and line 6.

It is possible that called procedure is connected to several other agents. Moreover, it is possible that any caller-agent has several communication instructions (**`in`** or **`out`**) that called that procedure at the different values of the program counter. Therefore, different variables could be sent. Therefore, the caller-agent and the value of its program counter has to be identified dynamically during the calculation of the **LTS**, cf. lines 7 up to 17. Fortunately, at the particular system state, there is an only one such agent with the given value of the program counter (It is impossible that several agents called the procedure due to the mutual-exclusion property of the passive agents' procedures). For each such case there is generated a dedicated predicate condition assuring that the system is in the expected state, cf. line 15, and the updated value of the system state regarding the caller-agent is then set, cf. line 16. The caller agent state is modified, cf. line 12. The caller-agent's mode is set to **F**inished, its program counter set to 0, its context list is cleared and the values of the caller-agent's variables are left unmodified, in the case that the active caller-agent's instruction is the last instruction. Otherwise, cf. line 10, caller-agent's mode is unmodified, its program counter is set to the value pointing to the next instruction, from its context list, the information of a calling the procedure is removed, and its variables are left unmodified.

```
1  if the agent is active then
2    newState ← "(F,0,empty,pv{index})"
3    newSystemState ← newSystemState with newState at index
4    print " = {newSystemState}"
5  else
6    newState ← "init_{agentName} pv{index}"
7    for each caller−agent and its pc then
8      callerNextPc ← next pc's value of caller−agent's pc
9      if caller−agent's pc is not LAST then
10       newCallerState ← "(am{callerIndex},{callerNextPc},delete (CProc {proceduralPortName} ci{callerIndex},pv{callerIndex})"
11     else
12       newCallerState ← "(F,0,empty,pv{callerIndex})"
13     end
14     newSystemState ← newSystemState with newState at index and with newCallerState at callerIndex
15     callingCondition ← "member (CProc {proceduralPortName}) ci{callerIndex} && pc{callerIndex} == {callerPc}"
16     print " | {callingCondition} = {newSystemState}"
17   end
18 end
```

Algorithm 4.8: Function fire generation algorithm – transition TExit

4.4.4 Transition TIn

The result of the execution of the transition **TIn** on the agent state depends on the type of the agent. The algorithm is presented in Algorithm 4.9.

When the system executes the transition **TIn**, that is related to an active agent that executes its instruction **`in`**, then the agent suspends its execution and waits for the peer agent to finalize the initialized communication. In terms of the agent's state, cf. line 3, it means that the agent's mode changes to **W**aiting, its program counter is left unmodified, a context item of `CIn` is inserted to its context list, and the values of the agent's variables are left unmodified.

When the system executes the transition **TIn**, that is related to a passive agent that executes its instruction **`in`**, then there are two distinct cases based on the type of the port used as the argument of the instruction

`in`.

When the port is a non-procedural one, the situation is similar to the active agent's case mentioned above, but this time the active context-agent that directly or indirectly executes the procedure is suspended. In terms of the agent's states, cf. line 8, it means that the passive agent's mode is left unmodified, its program counter is left unmodified, a context item of `CIn` is inserted to its context list, and the values of agent's variables are left unmodified. Additionally, the new context active agent's state is as follows, cf. line 11: the agent's mode changes to **W**aiting, its program counter is left unmodified, its context list is left unmodified, and the values of agent's variables are left unmodified.

On the other hand, when the port used by the instruction **`in`** is the port of the procedure being executed, the transition signifies the situation when the procedure completes exchanging value through the communication channel. The value received from the agent that directly called the procedure. The agent that directly called the procedure is called a *caller-agent*. In terms of the agent's states, cf. lines 20 up to 28, it means that the passive agent's mode is left unmodified, its program counter is advanced to the value indicating next instruction to-be-executed, its context list is left unmodified, and the values of the agent's variables are updated by a rewrite rule. The variable that is the argument of the instruction **`in`** has its value set to the value sent by the caller-agent in the case of value exchanging communication, or the agent's variables are left unmodified in the case of the signal exchange. The state of the caller-agent is left unmodified.

Like in the case of the instruction **`exit`** 4.4.3, it is possible that the called procedure is connected to several other agents. Therefore, the caller-agent and the value of its program counter has to be identified dynamically during the calculation of the **LTS**. For each such case, cf. line 30, there is generated a dedicated condition assuring that the system is in the expected state and next value of a system state is then set appropriately.

```
1  portName ← name of the port that is used by the executed instruction
2  if the agent is active then
3    newState ← "(W,pc{index},insert (CIn {portName}) ci{index},pv{index})"
4    newSystemState ← newSystemState with newState at index
5    print " = {newSystemState}"
6  else
7    if portName is non−procedural then
8      newState ← "(am{index},pc{index},insert (CIn {portName}) ci{index},pv{index})"
9      procedure ← currently executed procedure
10     for each context agent that is able to execute the procedure then
11       newContextAgentState ← "(W,pc{index},ci{index},pv{index})"
12       condition = "isWithinContextOf_{contextAgentName} {procedureName} s"
13       newSystemState ← newSystemState with newState at index and with newContextAgentState at contextAgentIndex
14       print " | {condition} = {newSystemState}"
15     end
16   else
17     procedure ← currently executed procedure
18     for each callerAgent that is able to directly execute the procedure then
```

```
            for each callerPc at which corresponding out instruction is executed by callerAgent then
                if communication sends values then
                    expression ← expression related to out instruction of callerAgent at callerPc
                    parsedExpression ← expression rewrite, cf. Sec. 4.4.4
                    varIndex ← index of variable used by in instruction
                    values ← newVariables of index with parsedExpression at varIndex
                else
                    values ← "pv{index}"
                end
                newState ← "(am{index},{nextPc},ci{index},{values})"
                callerIndex ← index of caller agent
                condition = "member (CProc {procedureName}) ci{callerIndex} && pc{callerIndex} == {callerPc}"
                newSystemState ← newSystemState with newState at index
                print " | {condition} = {newSystemState}"
            end
        end
    end
end
```

Algorithm 4.9: Function fire generation algorithm – transition TIn

4.4.5 Transition TInAP

The transition **TInAP** is related to establishing communication between the active agent and the passive agent's procedure. The algorithm is presented in Algorithm 4.10.

The system executing transition `TInAP A_p P_q x` is related only to the active agent A that executes its instruction **`in`** `p` and establishes successfully connection with the passive agent's procedure P_q. The execution of the active agent does not progress until the procedure that was called is finished. The value representing transition already has all the necessary information to pinpoint the procedure that was called.

In terms of the agent's state, cf. line 2, it means that the active agent's mode is left unmodified, its program counter is left unmodified, a context item of `CProc` is inserted to its context list, and the values of the agent's variables are left unmodified.

The state of the called passive agent is modified as follows, cf. line 5: the mode is changed to **T**aken, the program counter is set the value program counter of the first instruction of the called procedure, the context list is cleared, also, its variables are left unmodified.

```
1 calledPortName ← name of the called procedural port
2 newState ← "(am{index},ci{index},insert (CProc {calledPortName}) ci{index},pv{index})"
3 calledIndex ← index of the called agent
4 fistPcOfCalledProcedure ← program counter value of the first instruction of the called procedure
5 calledAgentState ← "(T,{fistPcOfCalledProcedure},empty,pv{calledIndex})"
6 newSystemState ← newSystemState with newState at index and with calledAgentState at calledIndex
7 print " = {newSystemState}"
```

Algorithm 4.10: Function fire generation algorithm – transition TInAP

4.4.6 Transition TInPP

The transition **TInPP** is related to establishing the communication between a passive agent (caller-agent) and another-passive agent's procedure (called procedure). The change of the system state is very similar to the change of the system after the transition **TInAP**, cf. Sec. 4.4.5. However, in this case, the caller-agent is a passive one. The algorithm is presented in Algorithm 4.11.

The system execution of the transition `TInPP P_p Q_q x` is related only to the passive agent P that executes its instruction **`in`** `p` and establishes successfully connection with the passive agent's procedure Q_q. The execution of the caller-agent does not progress until the called procedure is finished. The value representing transition already has all the necessary information to pinpoint the procedure that was called.

In terms of the agent's state, cf. line 2, it means that the caller-agent's mode is left unmodified, its program counter is left unmodified, a context item of `CProc` is inserted to its context list, and the values of the agent's variables are left unmodified.

The state of the called passive agent is modified as follows, cf. line 5: the mode is changed to **T**aken, the program counter is set to the value of program counter of the first instruction to be executed for the called procedure, the context list is cleared of any values, and its variables are left unmodified.

```
1 calledPortName ← name of the called procedural port
2 newState ← "(am{index},ci{index},insert (CProc {calledPortName}) ci{index},pv{index})"
3 calledIndex ← index of the called agent
4 fistPcOfCalledProcedure ← program counter value of the first instruction of the called procedure
5 calledAgentState ← "(T,{fistPcOfCalledProcedure},empty,pv{calledIndex})"
6 newSystemState ← newSystemState with newState at index and with calledAgentState at calledIndex
7 print " = {newSystemState}"
```

Algorithm 4.11: Function fire generation algorithm – transition TInPP

4.4.7 Transition TInF

The transition **TInF** is related to the completion of the communication between two active agents under the condition that the active peer agent had already initiated the communication, cf. Sec. 4.4.12. The active agent that previously executed its instruction **`out`** will be called **initializer agent**, and the agent that now finalizes communication by execution of the instruction **`in`** will be called **finalizer agent**. The algorithm is presented in Algorithm 4.12.

The system execution of the transition **TInF** is related only to the active agent that executes the instruction **`in`**, that successfully had established a connection with another active agent. When the transition

TOut had already been executed by the system, cf. Sec. 4.4.4, the initializer agent has already initiated the communication by the execution of its instruction **`out`** then the initializer agent is in the **W**aiting mode, and it waits for finalization of the communication. The execution of the transition **TInF** related to the finalizer agent wakes the initializer agent and sends the value through the communication channel from the initializer agent toward the finalizer agent.

In terms of the agent's state, cf. lines 11 up to 15, it means that the finalizer agent's mode is left unmodified, or it is changed to **F**inished in the case of the corresponding the instruction **`in`** is the last instruction, its program counter is advanced to the value of program counter of next instruction, or it is set to 0 in the case of the corresponding the instruction **`in`** is the last instruction, its context list is left unmodified, or it is cleared in the case of the corresponding the instruction **`in`** is the last instruction, and the values of the agent's variables are changed in the way that the variable that is the argument of the instruction **`in`** has set value to the value sent by the initializer agent in the case of value exchanging communication, or the agent's variables are left unmodified in the case of a signal exchange.

The state of the initializer agent is modified as follows, cf. lines 17 up to 21: its mode is changed to e**X**ecuting (Running), or it is changed to **F**inished in the case of the corresponding instruction **`out`** is the last instruction, its program counter is changed to the value of program counter of its next instruction, or it is set to 0 in the case of the corresponding instruction **`out`** is the last instruction, the entries COut and CTimer are simultaneously removed from its context list, or its context list is cleared in the case of the corresponding instruction **`out`** is the last instruction, finally, the value of its variables is left unmodified.

Like in the case of the instruction **`exit`**, cf. Sec. 4.4.3, it is possible that the finalizer agent is connected to several initializer agents. Therefore, the initializer agent and the value of its program counter has to be identified dynamically during the calculation of the **LTS**. If there are several such values, for each of them it is generated a separate transition **TInF** by the function `enable`. Additionally, there is generated a dedicated predicate condition assuring that the system is in the expected state, and the next value of a system state is then set appropriately.

```
1  portName ← name of the port that is used by the executed instruction
2
3  for each initializerPc of initializer agent where there is a matching out instruction ("matching" means the port's name is the same as
       ↪ specified in transition and sent value type is the same as in the instruction)
4    if communication is just a signal exchange then
5      pv ← "pv{index}"
6    else
7      sentValue ← variable sent by output instruction of the initializer agent rewrite
8      varIndex ← index of variable used by in instruction
9      pv ← newVariables of index with sentValue at varIndex
10   end
11   if nextPc is LAST then
12     newState ← "(F,0,empty,{pv})"
```

```
    else
        newState ← "(am{index},{nextPc},ci{index},{pv})"
    end
    initializerNewPc ← next pc after execution of instruction at initializerPc
    if initializerNewPc is LAST then
        initializerAgentState ← "(F,0,empty,pv{initializerIndex})"
    else
        initializerAgentState ← "(X,{initializerNewPc},deleteMany [COut initializerPort, selectTimer {initializerPc} ci{initializerIndex}] ci{
            ↪ initializerIndex},pv{initializerIndex})"
    end
    newSystemState ← newSystemState with newState at index and with initializerAgentState at initializerIndex
    condition ← "pc{initializerIndex} == {initializerPc}"
    print " | {condition} = {newSystemState}"
end
```

Algorithm 4.12: Function fire generation algorithm – transition TInF

4.4.8 Transition TJump

The result of the execution of the transition **TJump** on the agent's state does not depend on the type of the agent. The algorithm is presented in Algorithm 4.13.

The execution of the transition **TJump** by the system affects only the program counter of the agent that executed the instruction **`jump`**. The value of the program counter is advanced to the value referenced by the label used in the argument of the instruction **`jump`**. The label points to another instruction. The only restrictions posed on the label is that it has to be already defined for some instruction and that instruction has to be present in the currently executed procedure if the agent is passive or the main code block if the agent is active. The jumps between different procedures are forbidden, as well as the jumps between code of different agents.

In terms of the agent's state, cf. line 2, execution of instruction **`jump`** by the agent (both passive and active) means that the agent's mode is left unmodified, its program counter is set to the value of the program counter indicated by the label – the argument of the instruction, its context list is left unmodified, and the values of the agent's variables are left unmodified.

```
1  jumpToPc ← nextPc
2  newState ← "(am{index},{jumpToPc},ci{index},pv{index})"
3  newSystemState ← newSystemState with newState at index
4  print " = {newSystemState}"
```

Algorithm 4.13: Function fire generation algorithm – transition TJump

4.4.9 Transition TLoop

The result of the execution of the transition **TLoop** on the agent's state depends on the type of the agent. The algorithm is presented in Algorithm 4.14.

The execution of **TLoop** is related to checking the condition of the instruction **`loop`**. If the condition is resolved to be true, then the next instruction to be executed is the first instruction within the loop block of code. Otherwise, the next instruction to be executed is the first instruction outside the loop block of code. Therefore, the transition affects, in most cases, only agent program counter. The only exception to this rule is when **`loop`** is the last statement to be executed for the active agent. Then, if the condition is resolved to false, the active agent state is changed to **F**inished.

In terms of the agent's state, execution of the instruction **`loop`** by the agent (both passive and active) generates two possible states, differentiated dynamically during the generation of LTS by the condition of the instruction **`loop`**. The first state is reachable when the loop condition is resolved to be true, and therefore, cf. line 9, the agent's mode is left unmodified, its program counter is set to the value of program counter of the first instruction within the loop's block of code, its context list is left unmodified, and the values of the agent's variables are left unmodified.

The second state is reachable when the condition was resolved to be false. Then, cf. lines 10 up to 14, the agent's mode is left unmodified, or set to **F**inished, in the case when there is no instruction after the statement **`loop`**. I.e., the loop together with its block of code is the last instruction block. Agent program counter is set to the value of program counter of the first instruction after **`loop`** statement, or it is set to 0 in the case when **`loop`** is last statement. The agent's context list is left unmodified, or is cleared in the case when **`loop`** is last statement, and the values of the agent's variables are left unmodified unconditionally.

There is one optimization introduced when the instruction **`loop`** has no condition, i.e. it is an infinite loop, or the condition is the trivial value of **`True`**, cf. lines 2 up to 6. Then, there is generated only the first mentioned branch with no condition attached.

```
1  condtion ← condition expression of the corresponding loop instruction
2  if condition is just single true variable or there is no codition, e.g. loop() {...} then
3    newState ← "(am{index},{nextPc},ci{index},pv{index})"
4    newSystemState ← newSystemState with newState at index
5    print " = {newSystemState}"
6  else
7    parsedConditionExpression ← replace all local variables in expression to be performed with corresponding pv{index}_{varindex}
8    otherwiseNextPC ← value of program counter of instruction to be exectued when loop condition is resolved to false
9    newStateWhenConditionIsMet ← "(am{index},{nextPc},ci{index},pv{index})"
10   if otherwiseNextPc is not LAST then
11     newStateOtherwise ← "(am{index},{otherwiseNextPc},ci{index},pv{index})"
12   else
13     newStateOtherwise ← "(F,0,empty,pv{index})"
14   end
```

```
 newSystemStateWhenConditionIsMet ← newSystemState *with* newStateWhenConditionIsMet *at* index
 *print* " | {parsedConditionExpression} = {newSystemStateWhenConditionIsMet}"
 newSystemStateOtherwise ← newSystemState *with* newStateOtherwise *at* index
 *print* " | otherwise = {newSystemStateOtherwise}"
*end*
```

Algorithm 4.14: Function fire generation algorithm – transition TLoop

4.4.10 Transition TLoopEvery

The result of the execution of the transition **TLoopEvery** on the agent's state does not depend on the type of the agent. The algorithm is presented in Algorithm 4.15.

The execution of **TLoopEvery** is similar to the case of the transition **TLoop**, cf. Sec. 4.4.9, related to the unconditional instruction **`loop`**. Additionally, the value `CTimer` is added to its context list that indicates the time left for new evaluation of the loop. The next instruction to be executed is the first instruction from the loop block of code.

In terms of the agent's state, cf. line 5, execution of the instruction **`loop every`** by the agent (both passive and active) alters the agent's state as follows: the agent's mode is left unmodified, its program counter is set to the value of program counter of the first instruction from the loop block of code, the value of CTimer is inserted into its context list, the time argument placed into CTimer is calculated dynamically based on the argument t of the instruction `loop (every t)` minus the time required to perform loop instruction itself. The time for execution of any instruction is provided by the helper function `duration`, cf. Sec. 4.5.1. The values of the agent's variables are left unmodified.

```
1 condtion ← condition expression of the corresponding loop instruction
2 timeUnits ← time units as stated *in* the corresponding loop(every time units) instruction
3 newCi ← "insert (CTimer {pc} ({timeUnits} – (duration {agentName} {pc}))) ci{index}"
4
5 newState ← "(am{index},{nextPc},{newCi},pv{index})"
6 newSystemState ← newSystemState *with* newState *at* index
7 *print* " = {newSystemState}"
```

Algorithm 4.15: Function fire generation algorithm – transition TLoopEvery

4.4.11 Transition TNull

The result of the execution of the transition **TNull** on the agent's state depends on the type of the agent. The algorithm is presented in Algorithm 4.16.

The execution of the transition **TNull** by the system affects only the program counter of the agent executing the instruction **`null`** in most cases, with two exceptions. The standard meaning of the instruction is that it is executed with no side effects, and the execution progresses to the next instruction.

In terms of the agent's state, cf. line 20, execution of the instruction **`null`** by the agent (both passive and active) means that the agent's mode is left unmodified, its program counter is set to the value of program counter of the next instruction, its context list is left unmodified, and the values of the agent's variables are left unmodified.

The first exception concerns only active agents when the instruction **`null`** is the last one of the active agent's block of code. Then, the active agent transitions to **F**inished state.

In terms of the agent's state, cf. line 16, execution of instruction **`null`** by an active agent means that the agent's mode is changed to **F**inished, its program counter is set to 0, its context list is cleared, and the values of the agent's variables are left unmodified.

The second exception concerns both active and passive agent, and it is when the instruction is the last instruction of the **`loop every`** statement. Then after execution of **`null`** instruction, cf. line 3, the further progress is suspended until the next evaluation of the **`loop every`** block of code. Therefore, if a passive agent executes the instruction **`null`**, the program counter is advanced to the program counter value of **`loop every`** instruction, and further execution is suspended.

In terms of the agent's state, cf. line 7, execution of instruction **`null`** by a passive agent means that the agent's mode is left unmodified, its program counter is advanced to the next instruction, which is **`loop`** ↪ **`every`**, its context list is left unmodified, and the values of the agent's variables are left unmodified.

Additionally, for every possible context active agent executing the code of the procedure, the state of that agent is conditionally modified too. For each such a case, cf. lines 9 up to 13, there is generated a dedicated condition assuring that the system is in the state that the to-be-modified agent state is executing the procedure. The context active agent's state is then modified as follows: its mode is changed to **W**aiting, its program counter is left unmodified, its context list is left unmodified, and the values of the agent's variables are left unmodified.

When an active agent executes **`null`**, then the agent's mode is changed to **W**aiting, its program counter is advanced to the next instruction, its context list is left unmodified, and the values of the agent's variables are left unmodified.

```
1  if executed instruction is the last one within the loop every's block then
2    if the agent is active then
3      newState ← "(W,{nextPc},ci{index},pv{index})"
4      newSystemState ← newSystemState with newState at index
5      print " = {newSystemState}"
6    else
7      newState ← "(am{index},{nextPc},ci{index},pv{index})"
8      procedure ← currently executed procedure
9      for each context agent that is able to execute procedure then
10       newContextAgentState ← "(W,pc{index},ci{index},pv{index})"
11       newSystemState ← newSystemState with newState at index and with newContextAgentState at contextAgentIndex
12       print " | {condition} = {newSystemState}"
```

```
        end
    end
else if agent is active and executed instruction is the last one then
    newState ← "(F,0,empty,pv{index})"
    newSystemState ← newSystemState with newState at index
    print " = {newSystemState}"
else
    newState ← "(am{index},{nextPc},ci{index},pv{index})"
    newSystemState ← newSystemState with newState at index
    print " = {newSystemState}"
end
```

Algorithm 4.16: Function fire generation algorithm – transition TNull

4.4.12 Transition TOut

The result of the execution of the transition **TOut** on the agent's state depends on the type of the agent. The algorithm is very similar to algorithm for the transition **TIn**, cf. Sec. 4.4.4. However, in the case of passive agent executing `out` instruction having its procedural port as an argument, the direction of the variable passing is opposite. The algorithm is presented in Algorithm 4.17.

When an active agent executes the transition **TOut**, it corresponds to the execution of the instruction `out` by the agent. The agent suspends its execution and waits for the peer to finalize its communication. In terms of the agent's state, cf. line 3, it means that the agent's mode changes to **W**aiting, its program counter is left unmodified, a context item of `COut` is inserted to its context list, also, the values of the agent's variables are left unmodified.

When a passive agent executes a **TOut** transition, there are two distinct cases based on the type of the port used in the `out` instruction.

When the port is a non-procedural one, the situation is similar to the active agent's case mentioned above but this time, the execution of the active context agent that directly or indirectly executes the procedure is suspended. In terms of the agent's state, cf. line 9, it means that the passive agent's mode is left unmodified, its program counter is left unmodified, a context item of `COut` is inserted to its context list, and the values of agent's variables are left unmodified. Additionally, the new context active agent's state is as follows, cf. lines 11 up to 16: the agent's mode changes to **W**aiting, its program counter is left unmodified, its context list is left unmodified, and the values of agent's variables are left unmodified. The context agent has to be selected dynamically during the LTS generation and therefore special conditions are generated, cf. line 13.

On the other hand, when the port used by `out` instruction is the port of the procedure being executed, the transition signifies the situation when the procedure completes exchanging value through the communication channel by sending a value to the agent that directly called the procedure. The agent that directly called the procedure is called a *caller-agent*. In terms of the agent's states, cf. line 9, it means that the passive agent's mode is left unmodified, its program counter is advanced to the value indicating next instruction to-be-executed, its context list is left unmodified, also, the values of the agent's variables are left unmodified.

The state of the caller-agent is modified as follows, cf. lines 23 up to 31: its mode is left unmodified, its program counter is left unmodified, its context list is left unmodified, also, the values of agent's variables are changed in the way that the variable that is the argument of its corresponding instruction **`in`** has set value to the value sent by the **`out`** instruction from the called procedure in the case of the value exchanging communication alternatively, the agent's variables are left unmodified in the case of the signal exchange.

Like in the case of the instruction **`exit`**, cf. Sec. 4.4.3, it is possible that the called procedure is connected to several caller-agents. Therefore, the caller-agent and the value of its program counter has to be identified dynamically during the calculation of the **LTS**. For each such a case, cf. line 32, there is generated a dedicated condition assuring that the system is in the expected state and next value of a system state is then set appropriately.

```
1  if the agent is active then
2    portName ← name of the port that is used by the executed command
3    newState ← "(W,pc{index},insert (COut {portName}) ci{index},pv{index})"
4    newSystemState ← newSystemState with newState at index
5    print " = {newSystemState}"
6  else
7    portName ← name of the port that is used by the executed command
8    if portName is non−procedural then
9      newState ← "(am{index},pc{index},insert (COut {portName}) ci{index},pv{index})"
10     procedure ← currently executed procedure
11     for each context agent that is able to execute the procedure then
12       newContextAgentState ← "(W,pc{index},ci{index},pv{index})"
13       condition = "isWithinContextOf_{contextAgentName} {procedureName} s"
14       newSystemState ← newSystemState with newState at index and with newContextAgentState at contextAgentIndex
15       print " | {condition} = {newSystemState}"
16     end
17   else
18     procedure ← currently executed procedure
19     newState ← "(am{index},{nextPc},ci{index},pv{index})"
20     for each callerAgent that is able to directly execute the procedure then
21       for each callerPc at which corresponding out command is executed by callerAgent then
22         callerIndex ← index of context active agent
23         if communication sends values then
24           expression ← expression used by out instruction
25           parsedExpression ← expression rewrite
26           valueToBeAssigned ← index of the variable used by in instruction by callerAgent at its callerPc
27           values ← newVariables of callerIndex with parsedExpression at valueToBeAssigned
28           callerNewState ← "(am{callerIndex},pc{callerIndex},ci{callerIndex},{values})"
29         else
30           callerNewState ← "state{callerIndex}"
```

```
            end
            condition = "member (CProc {procedureName}) ci{callerIndex} && pc{callerIndex} == {callerPc}"
            newSystemState ← newSystemState with newState at index and with callerNewState at callerIndex
            print " | {condition} = {newSystemState}"
        end
      end
    end
end
```

Algorithm 4.17: Function fire generation algorithm – transition TOut

4.4.13 Transition TOutAP

The **TOutAP** transition is related to establishing communication between the active agent and the passive agent's procedure. The algorithm is the same as of **TInAP** transition 4.4.5 because the direction of the communication matters only when the passive agent reads from its procedural port, and in this case only communication is established. The algorithm is presented in Algorithm 4.18.

Only an active agent can execute a **TOutAP** transition. The execution of the active agent does not progress until the procedure that was called is finished. The value representing transition possesses all the necessary information to pinpoint the procedure that was called.

In terms of the agent's state, it means that the active agent's mode is left unmodified, its program counter is left unmodified, a context item of `CProc` is inserted to its context list, and the values of the agent's variables are left unmodified.

The state of the called passive agent is modified as follows: the mode is changed to **T**aken, the program counter is set to the program counter value of the first instruction to be executed for the given procedure, the context list is cleared, and its variables are left unmodified.

```
calledPortName ← name of the called procedural port
newState ← "(am{index},ci{index},insert (CProc {calledPortName}) ci{index},pv{index})"
calledIndex ← index of the called agent
fistPcOfCalledProcedure ← program counter value of the first instruction of the called procedure
calledAgentState ← "(T,{fistPcOfCalledProcedure},empty,pv{calledIndex})"
newSystemState ← newSystemState with newState at index and with calledAgentState at calledIndex
print " = {newSystemState}"
```

Algorithm 4.18: Function fire generation algorithm – transition TOutAP

4.4.14 Transition TOutPP

The **TOutPP** transition is related to establishing the communication between the passive agent (caller-agent) and another passive agent's procedure (called procedure). The change of the system state is very similar to the change of the system after **TOutAP** 4.4.13 transition but in this case, the caller-agent is a passive one.

On the other hand, the algorithm is the same as of **TInPP** transition 4.4.6 because the direction of the communication matters only when the called passive agent reads from its procedural port and in this case only communication is established. The algorithm is presented in Algorithm 4.19.

Only a passive agent can execute a **TOutPP** transition. The execution of the caller-agent does not progress until the called procedure is finished. The value representing transition possesses all the necessary information to pinpoint the procedure that was called.

In terms of the agent's state, it means that the caller-agent's mode is left unmodified, its program counter is left unmodified, a context item of `CProc` is inserted to its context list, and the values of the agent's variables are left unmodified.

The state of the called passive agent is modified as follow: the mode is changed to **T**aken, the program counter is set to the value of program counter of the first command to be executed for the called procedure, the context list is cleared of any values, and its variables are left unmodified.

```
* only passive agent
calledPortName ← name of the called procedural port
newState ← "(am{index},ci{index},insert (CProc {calledPortName}) ci{index},pv{index})"
calledIndex ← index of the called agent
fistPcOfCalledProcedure ← program counter value of the first instruction of the called procedure
calledAgentState ← "(T,{fistPcOfCalledProcedure},empty,pv{calledIndex})"
newSystemState ← newSystemState with newState at index and with calledAgentState at calledIndex
print " = {newSystemState}"
```

Algorithm 4.19: Function fire generation algorithm – transition TOutPP

4.4.15 Transition TOutF

The **TOutF** transition is related to the completion of the communication between two active agents under the condition that the communication was already initiated by the peer active agent. The active agent that previously executed the instruction `in` will be called **initializer agent**, and the agent that now finalizes communication by execution of instruction `out` will be called **finalizer agent**. The algorithm is similar to the algorithm presented for the **TInF** transition 4.12 with the change of the direction of variable exchange. The algorithm is presented in Algorithm 4.20.

Only an active agent can execute a **TOutF** transition. When in the system was already executed **TOut** transition (cf. 4.4.4) by an active agent, the initializer agent has already initiated the communication by execution of its `in` instruction then the agent is in the **W**aiting mode, and it waits for finalization of the communication. The execution of the transition **TOutF** by the finalizer agent wakes the initializer agent and sends the value through the communication channel from the finalizer agent toward the initializer agent.

In terms of the agent's state, it means that the finalizer agent's mode is left unmodified or it is changed to **F**inished in the case of the corresponding `out` instruction is the last instruction, its program counter is changed to the value of program counter of the next instruction or is set to 0 in the case of the corresponding **`out`** instruction is the last instruction, its context list is left unmodified, or is cleared in the case of the corresponding **`out`** instruction is the last instruction, and the values of the agent's variables are changed in the way that the variable that is the argument of the **`out`** instruction has its value set to the value sent by the initializer agent in the case of value exchanging communication or the agent's variables are left unmodified in the case of a signal exchange.

The state of the initializer agent is modified as follows: its mode is changed to e**X**ecuting (Running), or is changed to **F**inished when the instruction **`in`** is the last one, its program counter is changed to the value of program counter of its next instruction, or is set to 0 when **`in`** instruction is the last one, the entries `COut` and `CTimer` are simultaneously removed from its context list or its context list is cleared in the case of the corresponding **`in`** instruction is the last instruction, finally, the value of its variables is left unmodified.

Like in the case of instruction **`exit`** 4.4.3, it is possible that the finalizer agent is connected to several initializer agents. Therefore, the initializer agent and the value of its program counter has to be identified dynamically during the calculation of the **LTS**. If there are several such agents for each of them, it is generated a separate **TOutF** transition by enable function. Therefore, the execution of a particular **TOutF** transition has to consider only the different values of the program counter of the initializer agent. For each such a case, there is generated a dedicated condition assuring that the system is in the expected state and the next value of a system state is then set appropriately.

```
portName ← name of the port that is used by the executed command

for each initializerPc of initializer agent when there is matching in instruction ("matching" means the port name is the same as specified in
    ↪ the given transition and the sent value type is the same as in out instruction)
  initializerIndex ← index of initializer agent
  if communication is just a signal then
    initializerPv ← "pv{initializerIndex}"
  else
    sentValue ← expression used by out instruction of the finalizer agent
    parsedExpression ← sentValue rewrite
    varIndex ← index of variable used by in instruction of the initializer agent
    initializerPv ← newVariables of initializerIndex with parsedExpression at varIndex
  end
  if nextPc is LAST then
    newState ← "(F,0,empty,pv{index})"
  else
    newState ← "(am{index},{nextPc},ci{index},pv{index})"
  end
  initializerNewPc ← next pc after execution of instruction at calledPc
  if initializerNewPc is LAST then
    initializerAgentState ← "(F,0,empty,{initializerPv})"
  else
    initializerAgentState ← "(X,{initializerNewPc},deleteMany [CIn {calledPort}, selectTimer {initializerPc} ci{initializerIndex}] ci{
            ↪ initializerIndex},{initializerPv})"
    end
    newSystemState ← newSystemState with newState at index and with initializerAgentState at initializerIndex
    condition ← "pc{initializerIndex} == {initializerPc}"
    print " {condition} = {newSystemState}"
end
```

Algorithm 4.20: Function fire generation algorithm – transition TOutF

4.4.16 Transition TSelect

The result of the execution of the transition **TSelect** on the agent's state depends on the type of the agent to some extent. The algorithm is presented in Algorithm 4.21.

The execution of the transition **TSelect** by the agent affects only its program counter, with one exception. The processor evaluates sequentially all guarded branches to find the first branch that condition evaluates to true (so-called **open branch**). The branches are evaluated in order of appearance in the code layer. Then, execution of the agent is continued from the first instruction of the open branch. When none of the conditions is true, then execution is continued from the next instruction after the select statement (select instruction and all branches code blocks). In the last case, if there is no such an instruction (it is possible only in the case of an active agent) the agent finishes its execution.

In terms of the agent's state if there is at least one open branch, execution of instruction `select` by an agent (both passive and active) means that the agent's mode is left unmodified, its program counter is set to the value of program counter of the first instruction of the first open branch, its context list is left unmodified, and the values of the agent's variables are left unmodified. Evaluation of conditions is achieved by providing condition before modification of the agent's state in the fire function implementation.

When there is no open branch, but `select` is not the last statement of an active agent the agent's mode is left unmodified, its program counter is set to the value of program counter of the first instruction after the `select` statement, its context list is left unmodified, and the values of the agent's variables are left unmodified.

The last case is when there is no open branch and there is no instruction after the `select` statement – only active agents. Then the agent's mode is modified to **F**inished, its program counter is set to 0, its context list is cleared and set to the `empty` list, and the values of the agent's variables are left unmodified.

```
foreach branch of the select statement corresponding to the transition then
    newBranchPc ← pc of the first instruction inside the branch
    parsedConditionExpression ← replace all local variables in condition expression to be executed with corresponding pv{index}_{
        ↪ varindex}
    newBranchState ← "(am{index},{newBranchPc},ci{index},pv{index})"
    newSystemStateBranchConditionMet ← newSystemState with newBranchState at index
    print " | {parsedConditionExpression} = {newSystemStateBranchConditionMet}"
```

```
end
newPcOtherwise ← pc of the next instruction after all select step instructions
if newPcOtherwise is LAST then
    newStateOtherwise ← "(F,0,empty,pv{index})"
else
    newStateOtherwise ← "(am{index},{newPcOtherwise},ci{index},pv{index})"
end
newSystemStateOtherwise ← newSystemState with newStateOtherwise at index
print " | otherwise = {newSystemStateOtherwise}"
```

Algorithm 4.21: Function fire generation algorithm – transition TSelect

4.4.17 Transition TStart

The result of the execution of the transition **TStart** affects two agents' states one of which is an agent that executes **`start`** instruction (so-called **starter agent**) and the other is an active agent that is started (so-called **startee agent**). The algorithm is presented in Algorithm 4.22.

The execution of the transition **TStart** affects the starter agent's program counter and changes mode of the startee agent to running when the startee agent mode is **I**nit. There are two exceptions to that rule. One exception is when the instruction **`start`** is the last instruction of the starter agent (only in the case of an active agent), then the starter agent transitions to the finished state. The second exception is when the mode of the startee agent is not **I**nit, then no modification of the startee agent's state is applied.

In terms of the starter agent's state if the **`start`** instruction is not the last instruction of that agent, execution of the instruction by the agent (both passive and active) means that the agent's mode is left unmodified, its program counter is advanced to the next instruction, its context list is left unmodified, and the values of the agent's variables are left unmodified.

When **`start`** instruction is the last instruction of an active agent, the agent's mode is modified to **F**inished, its program counter is set to 0, its context list is cleared and set to the `empty` list, and the values of the agent's variables are left unmodified.

In terms of the startee agent's state if the agent is in the **I**nit mode, execution of the instruction by the starter agent means that the startee agent's mode is changed to e**X**ec, its program counter is set to 1, its context list is cleared, and the values of the agent's variables are left unmodified. The condition is posed by the fire function to assure that the startee agent is in the **I**nit mode.

When the startee agent's mode is other than **I**nit, no alteration of the startee agent's state happens.

```
if newPc is LAST then
    newState ← "(F,0,empty,pv{index})"
else
    newState ← "(am{index},{nextPc},ci{index},pv{index})"
end
```

```
starteeIndex ← index of startee agent
starteeState ← "(X,1,empty,pv{starteeIndex})"
newSystemStateNotRunning ← newSystemState with newState at index and with starteeState at starteeIndex
newSystemStateOtherwise ← newSystemState with newState at index
print " | am{starteeIndex} == I = {newSystemStateNotRunning}"
print " | otherwise = {newSystemStateOtherwise}"
```

Algorithm 4.22: Function fire generation algorithm – transition TStart

4.4.18 Transition STInAP

The result of the execution of the transition **STInAP** affects two agent's states, one of which is an active agent that executes the instruction `in` (so-called **caller-agent**) and the other is a passive agent that procedure is called (so-called **called agent**). The algorithm is presented in Algorithm 4.23.

Prior to the execution of the **STInAP** transition the caller-agent had been suspended in the **W**aiting mode. That situation had occurred because there was no available receiver of the communication before (e.g. the called passive agent was **T**aken by another agent). Now the system state has changed the called agent now is available and the system transition of **STInAP** signifies that the caller-agent was chosen by the scheduler, and it is awakened and calls the procedure of called agent.

In terms of the caller-agent's state, execution of the `in` instruction modifies the state of its state by the special function `awake` which will be described later on in Sec. 4.5.9. Essentially, in the case of an active agent, the function removes `CTimer` entry and replaces the `CIn` entry with the corresponding `CProc` entry in the caller's context list and changes the mode of the agent to e**X**cuting mode.

The called agent's state is modified as follows: the agent's mode is changed to **T**aken, its program counter is set to the value of program counter of the first instruction in the called procedure, its context list is cleared, and the values of the agent's variables are left unmodified.

The additional condition is asserted if the caller-agent is actually able to be awakened by the special function `isAwakableAgentName` which will be described later on in Sec. 4.5.8.

```
calledPortName ← name of the called procedural port
newState ← "awake{agentName} state{index} {calledPortName}"
calledIndex ← index of the called agent
fistPcOfCalledProcedure ← program counter value of the first instruction of the called procedure
calledAgentState ← "(T,{fistPcOfCalledProcedure},empty,pv{calledIndex})"
newSystemState ← newSystemState with newState at index and with calledAgentState at calledIndex
print " | isAwakable{agentName} state{index} {calledPortName} = {newSystemState}"
```

Algorithm 4.23: Function fire generation algorithm – transition STInAP

4.4.19 Transition STInPP

The result of the execution of the transition **STInPP** affects three agents' states one of which is a passive agent that executes the instruction **`in`** (so-called **caller-agent**), the next one is an active agent that executes, possibly indirectly, procedure of the caller-agent (so-called **context-agent**), and the last one is a passive agent that procedure is called by the caller-agent (so-called **called agent**). The algorithm is presented in Algorithm 4.24.

Prior to the execution of the **STInPP** transition, the context agent that executes the procedure of the caller-agent had been suspended in the **W**aiting mode. That situation had occurred because there was no available receiver of the communication before (the called passive agent was **T**aken by another agent). Now the system state has changed and the to-be-called agent is now available. The system transition of **STInPP** signifies that the caller-agent was chosen by the scheduler, and its context agent is awakened and the caller-agent calls the procedure of called agent.

In terms of the caller-agent's state, execution of the **`in`** instruction modifies the state of its state by the special function `awake` which will be described later on in Sec. 4.5.9. Essentially, the function in the case of passive agent removes the entry `CTimer` and replaces the entry `CIn` with the corresponding `CProc` entry in the caller's context list but in contrast to the active agent case, it does not change the mode of the caller-agent.

The called agent's state is modified as follows: the agent's mode is changed to **T**aken, its program counter is set to the value of program counter of the first instruction in the called procedure, its context list is cleared, and the values of the agent's variables are left unmodified.

Additionally, the new context active agent's state is as follows, the agent's mode changes to e**X**cuting mode, its program counter is left unmodified, its context list is left unmodified, and the values of the agent's variables are left unmodified.

The correct context agent is chosen from all possible context agents that could call the procedure of caller-agent dynamically, at LTS graph generation, by the assertion made on the system state that the appropriate context agent is directly or indirectly executing the procedure. The special function of `isWithinContextOf_AgentName` is used as the condition 4.5.10.

```
calledPortName ← name of the called procedural port
newState ← "awake{agentName} state{index} {calledPortName}"

calledIndex ← index of the called agent
fistPcOfCalledProcedure ← program counter value of the first instruction of the called procedure
calledAgentState ← "(T,{fistPcOfCalledProcedure},empty,pv{calledIndex})"

procedure ← currently executed procedure
*for each* context agent that is able to execute procedure *then*
    newContextAgentState ← "(X,pc{index},ci{index},pv{index})"
    isWithinContext = "isWithinContextOf_{contextAgentName} {procedureName} s"
    isAwakable = "isAwakable{agentName} state{index} {calledPortName}"
    condition = "{isWithinContext} && {isAwakable}"
    newSystemState ← newSystemState *with* newState *at* index *and with* newContextAgentState *at* contextAgentIndex *and with*
        ↪ calledAgentState *at* calledIndex
    *print* " | {condition} = {newSystemState}"
*end*
```

Algorithm 4.24: Function fire generation algorithm – transition STInPP

4.4.20 Transition STOutAP

The result of the execution of the transition **STOutAP** affects two agent's states, one of which is an active agent that executes instruction **`out`** (so-called **caller-agent**) and the other is a passive agent that procedure is called (so-called **called agent**). The algorithm is presented in Algorithm 4.25. The algorithm is the same as of **STInAP** transtion 4.4.18.

Prior to the execution of the **STOutAP** transition, the caller-agent had been suspended in the **W**aiting mode. That situation had occurred because there was no available receiver of the communication before (e.g. the called passive agent was **T**aken by another agent). Now the system state has changed the called agent now is available and the system transition of **STOutAP** signifies that the caller-agent was chosen by the scheduler, and it is awakened and calls the procedure of the called agent.

In terms of the caller-agent's state, execution of the **`out`** instruction modifies the state of its state by the special function `awake` which will be described later on in Sec. 4.5.9. Essentially, the function removes `CTimer` entry and replaces `COut` entry with the corresponding `CProc` entry in the caller's context list and changes the mode of the agent to e**X**cuting mode.

The called agent's state if modified as follows: the agent's mode is changed to **T**aken, its program counter is set to the value of program counter of the first instruction in the called procedure, its context list is cleared, and the values of the agent's variables are left unmodified.

The additional condition is asserted if the caller-agent is actually awakable by the special function `isAwakableAgentName` which will be described later on in Sec. 4.5.8.

```
calledPortName ← name of the called procedural port
newState ← "awake{agentName} state{index} {calledPortName}"
calledIndex ← index of the called agent
fistPcOfCalledProcedure ← program counter value of the first instruction of the called procedure
calledAgentState ← "(T,{fistPcOfCalledProcedure},empty,pv{calledIndex})"
newSystemState ← newSystemState *with* newState *at* index *and with* calledAgentState *at* calledIndex
*print* " | isAwakable{agentName} state{index} {calledPortName} = {newSystemState}"
```

Algorithm 4.25: Function fire generation algorithm – transition STInAP

4.4.21 Transition STOutPP

The result of the execution of the transition **STOutPP** affects three agent's states one of which is a passive agent that executes instruction **out** (so-called **caller-agent**), the next one is an active agent that executes, possibly indirectly, procedure of the caller-agent (so-called **context-agent**), and the last one is a passive agent that procedure is called by the caller-agent (so-called **called agent**). The algorithm is presented in Algorithm 4.26.

Prior to the execution of the **STOutPP** transition, the context agent that had been executing the procedure of the caller-agent had been suspended in the **W**aiting mode. That situation had occurred because there was no available receiver of the communication before (the called passive agent was **T**aken by another agent). Now the system state has changed and the to-be-called agent is now available. The system transition of **STOutPP** signifies that the caller-agent was chosen by the scheduler, and its context agent is awakened and the caller-agent calls the procedure of called agent.

In terms of the caller-agent's state, execution of the **out** instruction modifies the state of its state by the special function awake which will be described later on in Sec. 4.5.9. Essentially, the function in the case of passive agent removes CTimer entry and replaces COut entry with the corresponding CProc entry in the caller's context list but in contrast to the active agent case, it changes the mode of the context-agent.

The called agent's state is modified as follows: the agent's mode is changed to **T**aken, its program counter is set to the value of program counter of the first instruction in the called procedure, its context list is cleared, and the values of the agent's variables are left unmodified.

Additionally, the new context active agent's state is as follows, the agent's mode changes to e**X**cuting mode, its program counter is left unmodified, its context list is left unmodified, and the values of the agent's variables are left unmodified.

The correct context agent is chosen from all possible context agents that could call the procedure of caller-agent dynamically, at LTS graph generation, by the assertion made on the system state that the appropriate context agent is directly or indirectly executing the procedure. The special function of isWithinContextOf_AgentName is used as the condition 4.5.10.

```
calledPortName ← name of the called procedural port
newState ← "awake{agentName} state{index} {calledPortName}"

calledIndex ← index of the called agent
fistPcOfCalledProcedure ← program counter value of the first instruction of the called procedure
calledAgentState ← "(T,{fistPcOfCalledProcedure},empty,pv{calledIndex})"

procedure ← currently executed procedure
for each context agent that is able to execute procedure then
  newContextAgentState ← "(X,pc{index},ci{index},pv{index})"
  isWithinContext = "isWithinContextOf_{contextAgentName} {procedureName} s"
```

```
    isAwakable = "isAwakable{agentName} state{index} {calledPortName}"
    condition = "{isWithinContext} && {isAwakable}"
    newSystemState ← newSystemState with newState at index and with newContextAgentState at contextAgentIndex and with
        ↪ calledAgentState at calledIndex
    print " | {condition} = {newSystemState}"
end
```

Algorithm 4.26: Function fire generation algorithm – transition STOutPP

4.4.22 Transition STDelayEnd

The result of the execution of the transition **STDelayEnd** on the agent's state depends on the type of the agent. This is because the transition awakes suspended execution of the agent that had been executing the code directly on the processing unit. Therefore, for an active agent, it is that agent itself but for a passive agent, this is an active context-agent that had been executing the code of the procedure directly or indirectly. The transition complements the **TDelay** transition 4.4.1 because it finishes the period of suspension of the agent that had been executing the code.

The algorithm is presented in Algorithm 4.27. In terms of the agent's state, when execution is resumed after the instruction **delay** suspended the active agent and the instruction **delay** is not the last one then it means that the agent's mode is changed to e**X**ecuting, its program counter is advanced to the next instruction, a context item of `CTimeout` is removed from its context list, and the values of the agent's variables are left unmodified.

On the other hand when the instruction **delay** executed by an active agent is the last instruction then it means that the agent's mode is changed to **F**inished, its program counter is set to 0, its context list is cleared, and the values of the agent's variables are left unmodified.

In the case when a passive agent had executed an instruction **delay**, the agent's mode is left unmodified, its program counter is set to the value of next instruction to be executed, a context item of `CTimeout` is removed from its context list, and the values of the agent's variables are left unmodified. Additionally, for every possible context active agent executing the code of the procedure, the state of that agent is conditionally modified too. For each such a case, there is generated a dedicated condition assuring that the system is in the state that the to-be-modified agent state is executing the procedure. Then the mode of the context agent is changed to e**X**ecuting mode, the context agent's program counter is left unmodified, the context agent's context list is left unmodified, and values of its variables are left unmodified.

```
if the agent is active then
    if newPc is not LAST then
        newState ← "(X,{newPc},delete (CTimeout {pc}) ci{index},pv{index})"
        newSystemState ← newSystemState with newState at index
        print " = {newSystemState}"
    else
```

```
        newState ← "(F,0,empty,pv{index})"
        newSystemState ← newSystemState *with* newState *at* index
        *print* " = {newSystemState}"
    *end*
*else*
    newState ← "(am{index},{newPc},delete (CTimeout {pc}) ci{index},pv{index})"
    procedure ← currently executed procedure
    *for each* context agent that is able to execute procedure *then*
        newContextAgentState ← "(X,pc{index},ci{index},pv{index})"
        condition = "isWithinContextOf_{contextAgentName} {procedureName} s"
        newSystemState ← newSystemState *with* newState *at* index *and with* newContextAgentState *at* contextAgentIndex
        *print* " | {condition} = {newSystemState}"
    *end*
*end*
```

Algorithm 4.27: Function fire generation algorithm – transition STDelayEnd

4.4.23 Transition STInEnd

The result of the execution of the transition **STInEnd** on the agent's state depends on the type of the agent. This is because the transition awakes suspended execution of the agent that had been executing the code directly on the processing unit. Therefore, for an active agent, it is that agent itself but for a passive agent, this is an active context-agent that had been executing the code of the procedure directly or indirectly. The transition complements the **TIn** transition, cf. Sec. 4.4.4, but it is related only to the non-blocking instruction `in` because it finishes the period of suspension of the agent that had been executing the code directly on the processing unit. It means that the agent failed to establish communication successfully within a specified period. The time-out event is generated, and the agent starts processing the `fail` block of the non-blocking `in` statement.

The algorithm is presented in Algorithm 4.28. In terms of the agent's state, when execution is resumed after the non-blocking instruction `in` suspended the active agent and the non-blocking instruction `in` is not the last one then it means that the agent's mode is changed to e**X**ecuting, its program counter is advanced to the first instruction of the `fail` block or the first instruction after `in` statement when no such block was defined, context items of both `CIn` and `CTimer` are removed from its context list, and the values of the agent's variables are left unmodified.

On the other hand when the non-blocking `in` is the last statement of the active agent then it means that the agent's mode is changed to **F**inished, its program counter is set to 0, its context list is cleared, and the values of the agent's variables are left unmodified.

In the case when a passive agent had executed the non-blocking instruction `in`, the agent's mode is left unmodified, its program counter is set to the value of next instruction to be executed, context items of both `CIn` and `CTimer` are removed from its context list, and the values of the agent's variables are left unmodified. Additionally, for every possible context active agent executing the code of the procedure, the state of that agent is conditionally modified too. For each such a case, there is generated a dedicated condition assuring that the system is in the state that the to-be-modified agent state is executing the procedure. Then the mode of the context agent is changed to e**X**ecuting mode, the context agent's program counter is left unmodified, the context agent's context list is left unmodified, and values of its variables are left unmodified.

```
portName ← port name as an argument of non−blocking in instruction
failedPc ← pc of the first instruction from the fail block of non−blocking in statement

if the agent is active then
  if failedPc is not LAST then
    newState ← "(X,{failedPc},deleteMany [CIn {portName},CTimeout {pc}] ci{index},pv{index})"
    newSystemState ← newSystemState with newState at index
    print " = {newSystemState}"
  else
    newState ← "(F,0,empty,pv{index})"
    newSystemState ← newSystemState with newState at index
    print " = {newSystemState}"
  end
else
  newState ← "(am{index},{failedPc},deleteMany [CIn {portName},CTimeout {pc}] ci{index},pv{index})"
  procedure ← currently executed procedure
  for each context agent that is able to execute procedure then
    newContextAgentState ← "(X,pc{index},ci{index},pv{index})"
    condition = "isWithinContextOf_{contextAgentName} {procedureName} s"
    newSystemState ← newSystemState with newState at index and with newContextAgentState at contextAgentIndex
    print " | {condition} = {newSystemState}"
  end
end
```

Algorithm 4.28: Function fire generation algorithm – transition STInEnd

4.4.24 Transition STLoopEnd

The result of the execution of the transition **STLoopEnd** on the agent's state depends on the type of the agent. This is because the transition awakes suspended execution of the agent that had been executing the code directly on the processing unit. Therefore, for an active agent, it is that agent itself but for a passive agent, this is an active context-agent that had been executing the code of the procedure directly or indirectly. The transition complements the **TNull** transition 4.4.11, but restricted to the case when **`null`** instruction is the last instruction of the **`loop every`** block of code. It finishes the period of suspension of the agent that had been executing the code directly on the processing unit, and allows evaluation of the **`loop every`** in the next step.

The algorithm is presented in Algorithm 4.29. In terms of the agent's state, when execution of the active agent is resumed from instruction of the **`loop every`**, then it means that the agent's mode is changed to e**X**ecuting, its program counter is left unmodified, the context item of `CTimeout` is removed from its context list, and the values of the agent's variables are left unmodified.

In the case of execution of the procedure of passive agent, execution of context agent is resumed from instruction **`loop every`**, the agent's mode is left unmodified, its program counter is not modified, a context item of `CTimeout` is removed from its context list, and the values of the agent's variables are left unmodified. Additionally, for every possible context active agent executing the code of the procedure, the state of that agent is conditionally modified too. For each such a case, there is generated a dedicated condition assuring that the system is in the state that the to-be-modified agent state is executing the procedure. Then the mode of the context agent is changed to e**X**ecuting mode, the context agent's program counter is left unmodified, the context agent's context list is left unmodified, and values of its variables are left unmodified. The program counters are not modified as **`null`** instruction already adjusted program counter.

```
if the agent is active then
  newState ← "(X,pc{index},delete (CTimeout {pc}) ci{index},pv{index})"
  newSystemState ← newSystemState with newState at index
  print " = {newSystemState}"
else
  newState ← "(am{index},pc{index},delete (CTimeout {pc}) ci{index},pv{index})"
  procedure ← currently executed procedure
  for each context agent that is able to execute procedure then
    newContextAgentState ← "(X,pc{index},ci{index},pv{index})"
    condition = "isWithinContextOf_{contextAgentName} {procedureName} s"
    newSystemState ← newSystemState with newState at index and with newContextAgentState at contextAgentIndex
    print " | {condition} = {newSystemState}"
  end
end
```

Algorithm 4.29: Function fire generation algorithm – transition STLoopEnd

4.4.25 Transition STOutEnd

The result of the execution of the transition **STOutEnd** on the agent's state depends on the type of the agent. This is because the transition awakes suspended execution of the agent that had been executing the code directly on the processing unit. Therefore, for an active agent, it is that agent itself but for a passive agent, this is an active context-agent that had been executing the code of the procedure directly or indirectly. The transition complements the **TOut** transition 4.4.12 but related only to the non-blocking instruction **`out`** because it finishes the period of suspension of the agent that had been executing the code directly on the processing unit. It means that the agent failed to establish communication successfully within a specified period. The time-out event is generated, and the agent starts processing the **`fail`** block of the non-blocking **`out`** statement.

The algorithm is presented in Algorithm 4.28. In terms of the agent's state, when execution is resumed after the non-blocking instruction **`out`** suspended the active agent and the non-blocking instruction **`out`** is not the last one then it means that the agent's mode is changed to e**X**ecuting, its program counter is advanced to the first instruction of the **`fail`** block, or the first instruction after **`out`** statement when no such block was defined. Context items of both `COut` and `CTimeout` are removed from its context list, and the values of the agent's variables are left unmodified.

On the other hand when the non-blocking **`out`** is the last statement of the active agent, then it means that the agent's mode is changed to **F**inished, its program counter is set to 0, its context list is cleared, and the values of the agent's variables are left unmodified.

In the case when a passive agent had executed the non-blocking instruction **`out`**, the agent's mode is left unmodified, its program counter is set to the value of next instruction to be executed, context items of both `COut` and `CTimeout` are removed from its context list, and the values of the agent's variables are left unmodified. Additionally, for every possible context active agent executing the code of the procedure, the state of that agent is conditionally modified too. For each such a case, there is generated a dedicated condition assuring that the system is in the state that the to-be-modified agent state is executing the procedure. Then the mode of the context agent is changed to e**X**ecuting mode, the context agent's program counter is left unmodified, the context agent's context list is left unmodified, and values of its variables are left unmodified.

```
portName ← port name as an argument of the instruction of non−blocking out
failedPc ← pc of the first instruction from the fail block of the statement of non−blocking out

if the agent is active then
  if failedPc is not LAST then
    newState ← "(X,{failedPc},deleteMany [COut {portName},CTimeout {pc}] ci{index},pv{index})"
    newSystemState ← newSystemState with newState at index
    print " = {newSystemState}"
  else
    newState ← "(F,0,empty,pv{index})"
    newSystemState ← newSystemState with newState at index
    print " = {newSystemState}"
  end
else
  newState ← "(am{index},{failedPc},deleteMany [COut {portName},CTimeout {pc}] ci{index},pv{index})"
  procedure ← currently executed procedure
  for each context agent that is able to execute procedure then
    newContextAgentState ← "(X,pc{index},ci{index},pv{index})"
    condition = "isWithinContextOf_{contextAgentName} {procedureName} s"
    newSystemState ← newSystemState with newState at index and with newContextAgentState at contextAgentIndex
    print " | {condition} = {newSystemState}"
  end
```

```
end
```

Algorithm 4.30: Function fire generation algorithm – transition STOutEnd

4.4.26 Transition STTime

The execution of the transition **STTime** corresponds to the passage of time in the system when no other transition is possible to be executed. The situation has to be distinguished from deadlock case, as eventually, there is some transition to be enabled only after some time. The transition is calculated dynamically and is model independent. Therefore, its algorithm can be copied from configuration file.

4.5 Auxiliary functions

4.5.1 duration

The template of the `duration` function is generated in such a way that it assigns one unit time to each instruction duration of every agent. Each instruction can be later adjusted separately for every agent and every program counter value. The algorithm generating the duration function was presented in Listing 4.31. The example of generated duration function for the Producer-Consumer problem is presented in Algorithm 4.39.

```
for each agent in the system then
  agentName $gets$ name of agent
  for each pc of the agent then
    print "duration {agentName} {pc} = 1"
  end
end
```

Algorithm 4.31: Algorithm generating function providing the duration of each step

```
duration :: Agent -> Int -> Int
duration Producer 1 = 1
duration Producer 2 = 1
duration Producer 3 = 1
duration Buffer 1 = 1
duration Buffer 2 = 1
duration Buffer 3 = 1
duration Buffer 4 = 1
duration Buffer 5 = 1
duration Buffer 6 = 1
duration Consumer 1 = 1
duration Consumer 2 = 1
duration _ _ = 1
```

Listing 4.39: Example of generated duration function – Producer-Consumer problem

4.5.2 procFree

The `procFree` function is responsible for testing if the passive agent is not executing any remote procedure. This is a helper function used by `enable` function 4.3. In Listing 4.40, there was presented just a function signature. The reason behind this is that the body of the function does not depend on the model and therefore there is no dedicated algorithm to generate it, but its definition is copied from the configuration file.

The first and the only argument of the function represents the context list of any agent that is to-be-tested. The result is the Boolean value of **`True`** – the agent does not execute any remote procedure or **`False`** – the agent does execute a remote procedure.

```
procFree :: Set ContextInfo -> Bool
```

Listing 4.40: The type signature of the procFree helper function

4.5.3 xContext

The `xContext` function is responsible for testing if the context agent that is executing the passive agent's procedure is in the e**X**ecuting mode. This is a helper function used by `enable` function 4.3. In Listing 4.41, there was presented just a function signature. The reason behind this is that the body of the function does not depend on the model and therefore there is no dedicated algorithm to generate it, but its definition is copied from the configuration file.

The first argument of the function represents the agent index of the passive agent that the procedure is executed by the context agent. The second argument is the state of the system. Then the function searches context lists, following the calling path, eventually finding the active context agent state and its mode is tested if it is in running mode. The result is the Boolean value of **`True`** – the context agent is in e**X**ecuting mode, or **`False`** – the agent is in other mode (usually **W**aiting).

```
xContext :: Int -> State -> Bool
```

Listing 4.41: The type signature of the xContext helper function

4.5.4 wContext

The `wContext` function is responsible for testing if the context agent that executes the passive agent's procedure is in the **W**aiting mode. This is a helper function used by `enable` function 4.3. In Listing 4.42, there was presented just a function signature. The reason behind this is that the body of the function does not depend on the model and therefore there is no dedicated algorithm to generate it, but its definition is copied from the configuration file.

The first argument of the function represents the index of the passive agent that the procedure is executed within the context of the context agent. The second argument is the state of the system. Then, the function searches context lists, following the calling path, eventually finding the active context agent state and its mode is tested if it is in running mode. The result is the Boolean value of **`True`** – the context agent is in **W**aiting mode, or **`False`** – the agent is in other mode (usually e**X**ecuting).

```
wContext :: Int -> State -> Bool
```

Listing 4.42: The type signature of the wContext helper function

4.5.5 selectTimer

The `selectTimer` function is responsible for extracting `CTimer` value related to the given program counter value from the context list. This is a helper function used by `fire` function 4.4. In Listing 4.40, there was presented just a function signature. The reason behind this is that the body of the function does not depend on the model and therefore there is no dedicated algorithm to generate it, but its definition is copied from the configuration file.

The first argument of the function represents the program counter value of the searched `CTimer` value. The second argument is the context list that is to be searched. The result is the value of `CTimer` if it is present or the value of `CNone` otherwise.

```
selectTimer :: Int -> Set ContextInfo -> ContextInfo
```

Listing 4.43: The type signature of the selectTimer helper function

4.5.6 init

The **`init`** function populates the context list of the passive agent with all available procedures based on the current value of its local variables. This is a helper function used by `fire` function when the transition of `TExit` 4.4.3 is considered and by the `s0` function 4.5.7.

Actually, **`init`** is not a single function, but rather a family of functions that differ by the name and type of arguments required. The instances of the **`init`** functions are generated for a passive agent only. Every passive agent in the system has its own version of **`init`** function. The name of the specific version of the function is appended with the name of the agent. The only argument is a tuple of the variables of the considered passive agent, it differs between different agents as they have different variable set. The result of the function is the value of the state of the passive agent.

The algorithm generating the **`init`** function is presented in Algorithm 4.32. For every passive agent, there is generated a dedicated instance of the function. Firstly, the set of all procedures of the agent is split into two disjoint sets. The one is the set of always available procedures i.e. those with no guard expression.

The second is the set of guarded procedures. Then, from the guarded procedure set, the power set is calculated and later for every element of the power set an if-then case of the function is generated. Therefore, an element of power set is a list of guarded procedures. For every element of the power set, the condition is generated as a concatenated rewritten guard expressions joined with a logical and operator. The state of the agent is then as follows its mode is set to **W**aiting, its program counter is set to 0, its context list is populated with context info representing procedures from the element of the power set and always available procedures, and its variables are set to the value passed as the argument to the function.

```
for each passive agent in the system then
  agentName ← name of agent
  index ← index of the agent
  print "init_{agentName} pv{index}@(pv{index}_1, pv{index}_2,...)"

  allProcedures ← all procedure of the agent

  allwaysAvailableProcedures ← select unguarded procedures from allProcedures
  ciValues ← all procedures of allwaysAvailableProcedures replaced with corresponding "CIn {procedureName}" or "COut {
      ↪ procedureName}" value
  alwaysAvailableProceduresCiValues ← all ciValues concatenated with "," separator

  guardedProcedures ← select procedures with guards from allProcedures
  for each list of guarded procedures from $2^{guardedProcedures}$ order by their cardinality descending then
    parsedGuards ← guard rewrite
    conditions ← all parsedGuards concatenated with "&&" operator
    ciValues ← all procedures replaced with corresponding "CIn {procedureName}" or "COut {procedureName}" value
    proceduresCiValues ← all ciValues concatenated with "," separator
    print " | {conditions} = (W,0,fromList [{proceduresCiValues},{alwaysAvailableProceduresCiValues}],pv{index})"
  end
  print " | otherwise = (W,0,fromList [{alwaysAvailableProceduresCiValues}],pv{index})"
end
```

Algorithm 4.32: Algorithm generating *init* helper function

The example of the generated **`init`** function for the Producer-Consumer problem is presented in Listing 4.44. Based on the provided example, it is visible the generative aspect of the **`init`** function. The state when both procedures `put` and `get` of the `Buffer` agent are available is unreachable (first line). The condition of both `pv2_2` and **`not`** `pv2_2` is always false. In the general case, such conditions are unavoidable due to the NP-completeness of the Boolean satisfiability problem [47]. However, when the valuation of the local variables of the agent is known, i.e. at runtime, the evaluation of the expression is fast and does not pose any negative feedback on the runtime 6.8.

```
init_Buffer pv2@(pv2_1, pv2_2)
  | pv2_2 && not pv2_2 = (W,0,fromList [COut Buffer_get, CIn Buffer_put],pv2)
  | pv2_2 = (W,0,fromList [COut Buffer_get],pv2)
  | not pv2_2 = (W,0,fromList [CIn Buffer_put],pv2)
  | otherwise = (W,0,fromList [],pv2)
```

Listing 4.44: Example of init function – Producer-Consumer problem

That algorithm is simple and readable for a few guarded procedures in a single passive agent. If the passive agent would have more guarded procedures, number of conditions grows exponentially and the algorithm should be replaced with one of linear complexity. E.g. for each guarded procedure, there could be generated a list expression which is an empty list or a singleton list depending on the evaluation of the guard expression. Then the context list is a concatenation of all tiny lists. That approach would be more efficient in terms of evaluation complexity, but it would be less readable. In the future, one could provide a hybrid solution that chooses the linear algorithm after some threshold of number of guarded procedures is exceeded, otherwise the simpler algorithm could be used.

4.5.7 State initialization

The `s0` constant represents the initial system state. This constant is required to generate the LTS graph.

The definition of the `s0` constant takes advantage of the family of `s0` constants that differ by the name of the agent. The value of the family of `s0` represents the initial state of a particular agent.

The algorithm generating the `s0` constant was presented in Algorithm 4.33. For every agent in the system, the initial state of that agent is generated. There are three separate cases: the initial state of a passive agent, the initial state of an active agent that was set into the e**X**ecuting mode in the model, and the initial mode of an active agent is **I**nitial mode. When the initial state of a passive agent is generated it is used **`init`** function 4.5.6 to populate its context list with all procedures that are available based on the values of the agent. When the initial state of a running active agent is generated its mode is set to e**X**ecuting, its program counter is set to 1, its context list is cleared, and its variables are set based on the default values of local variables as was stated in the code layer. When the initial state of a non-running active agent is generated its mode is set to **I**nitial, its program counter is set to 0, its context list is cleared, and its variables are set based on the default values of local variables as was stated in the code layer.

The initial state of the whole system is the tuple of initial states of the particular agents.

```
for each agent in the system then
  agentName ← name of agent
  values ← create a parsed set tuple of values based on the default values of local variables of the agent
  if passive is active then
    print "s0_{agentName} = init_{agentName} {values}"
  else if agent is active and is in running mode then
    print "s0_{agentName} = (X,1,empty,{values})"
  else
    print "s0_{agentName} = (I,0,empty,{values})"
  end
end
```

```

print "s0 :: State"
print "s0 = (s0_FirstAgentName, s0_SecondAgentName, ...)"
```

Algorithm 4.33: Algorithm generating s0 constant (inital system state)

The example of the generated `s0` function for the Producer-Consumer problem is presented in Listing 4.45. As one can see the initial state of the `Producer` agent is the second case, the initial state of the `Buffer` agent is the first case, and the initial state of the `Consumer` agent is the third case of the algorithm.

```
s0_Producer = (X,1,empty,(0))

s0_Buffer = init_Buffer (0, False)

s0_Consumer = (I,0,empty,(0))

s0 :: State
s0 = (s0_Producer, s0_Buffer, s0_Consumer)
```

Listing 4.45: Example of s0 constant (inital system state) – Producer-Consumer problem

4.5.8 isAwakable

The `isAwakable` function checks whether the agent is in the state in which it can be awakened to establish communication with a given remote port. This is a helper function used by `fire` function when one of the transitions of `STInAP` 4.4.18, `STInPP` 4.4.19, `STOutAP` 4.4.20, or `STOutPP` 4.4.21 is considered.

Actually, `isAwakable` is a family of functions that differ by the name and type of the arguments required. The instance of the `isAwakable` function is generated for each agent separately, and the name of the specific version of the function is appended with the name of the agent. The first argument is the tuple that represents the agent's state and therefore, it differs between different agents because of the different set of local variables. The second argument represents the remote port that is owned by another agent and with which the communication is supposed to be established. The result of the function is the boolean value of **`True`** or **`False`**.

The algorithm generating the `isAwakable` function was presented in Algorithm 4.34. For every port of the considered agent (so-called **local port**), it explores all the connections that lead to distinct remote ports. For each pair of the local port and the remote port, the direction of the communication between them is asserted and based on that the correct condition restricting the state of the agent is generated.

The example of generated `isAwakable` function for the Producer-Consumer problem is presented in Listing 4.46.

```
for each agent in the system then
  agentName ← name of agent
  *print* "isAwakable{agentName} (am,pc,ci,pv) key"

  *for each* local port of agent by which communication can be initialized ordered by code order definition *then*
    *for each* distinct remote port that can be called by local port *then*
      *if* communication *with* called port is output one *then*
          condition ← "member (COut {localPortName}) ci && key == {remotePortName}"
      *else*
          condition ← "member (CIn {localPortName}) ci && key == {remotePortName}"
      *end*
      *print* " | {condition} = True"
    *end*
  *end*

  *print* " | otherwise = False"
*end*
```

Algorithm 4.34: Algorithm generating isAwakable helper function

```
isAwakableProducer (am,pc,ci,pv) key
  | member (COut Producer_push) ci && key == Buffer_put = True
  | otherwise = False

isAwakableBuffer (am,pc,ci,pv) key
  | otherwise = False

isAwakableConsumer (am,pc,ci,pv) key
  | member (CIn Consumer_pull) ci && key == Buffer_get = True
  | otherwise = False
```

Listing 4.46: Example of isAwakable function – Producer-Consumer problem

4.5.9 awake

The `awake` function complements the `isAwakable` 4.5.8 because it calculates the state of the agent after it is awaken and the communication with a given remote port is established. This is a helper function used by `fire` function when one of the transitions of `STInAP` 4.4.18, `STInPP` 4.4.19, `STOutAP` 4.4.20, or `STOutPP` 4.4.21 is considered.

Actually, `awake` is a family of functions that differ by the name and type of the required arguments. The instance of the `awake` function is generated for each agent separately, and the name of the specific version of the function is appended with the name of the agent. The first argument is the tuple that represents the agent's state and therefore, it differs between different agents because of the different set of local variables. The second argument represents the remote port that is owned by another agent and with which the communication is supposed to be established. The result of the function is the new state of the agent.

The algorithm generating the `awake` function was presented in Algorithm 4.35. For every port of the considered agent (so-called **local port**), it explores all the connections that lead to distinct remote ports. For each pair of the local port and the remote port, the direction of the communication between them is asserted and based on that the correct condition restricting the state of the agent is generated. For an active agent, the new state is returned that is as follows: the agent mode is changed to e**X**ecuting, the program counter value is left unmodified, the value of `CProc` in inserted to its context list and values of `CIn` or `COut`, and `CTimer` are removed from the context list, agent's values are left unmodified. The value of `CIn` or `COut` is chosen based on the type of instruction used at that specific value of the program counter. The value of proper value of `CTimer` is selected by the `selectTimer` helper function 4.5.5.

The example of generated `awake` function for the Producer-Consumer problem is presented in Listing 4.47.

```
for each agent in the system then
  agentName gets name of agent
  print "awake{agentName} s@(am,pc,ci,pv) key"
  for each instruction of in or out of the agent by which communication can be initialized ordered by code order definition then
    for each distinct remote port that can be called from local port directly then

      calledAgentName ← name of the agent that is the owner of called port
      if communication with called port is output one then
        condition ← "member (COut {localPortName}) ci && key == {remotePortName}"
      else
        condition ← "member (CIn {localPortName}) ci && key == {remotePortName}"
      end

      if communication with called port is output one then
        contextInfoCtor ← "COut"
      else
        contextInfoCtor ← "CIn"
      end

      if the agent is active then
      newState ← "(X,pc,add(CProc {remotePortName}) (deleteMany[{contextInfoCtor} {localPortName},selectTimer {pc} ci] ci),pv)"
      else
      newState ← "(am,pc,add(CProc {remotePortName}) (deleteMany[{contextInfoCtor} {localPortName},selectTimer {pc} ci] ci),pv)"
      end
      print " | {condition} = {newState}"
    end
  end

  print " | otherwise = s"
end
```

Algorithm 4.35: Algorithm generating awake helper function

```
awakeProducer s@(am,pc,ci,pv) key
  | member (COut Producer_push) ci && key == Buffer_put = (X,pc,insert (CProc Buffer_put) (deleteMany [COut Producer_push,(
        ↪ selectTimer 3 ci)] ci),pv)
  | otherwise = s

awakeBuffer s@(am,pc,ci,pv) key
  | member (COut Buffer_get) ci && key == Consumer_pull = (am,pc,insert (CProc Consumer_pull) (deleteMany [COut Buffer_get,(
        ↪ selectTimer 1 ci)] ci),pv)
  | member (CIn Buffer_put) ci && key == Producer_push = (am,pc,insert (CProc Producer_push) (deleteMany [CIn Buffer_put,(
        ↪ selectTimer 4 ci)] ci),pv)
  | otherwise = s

awakeConsumer s@(am,pc,ci,pv) key
  | member (CIn Consumer_pull) ci && key == Buffer_get = (X,pc,insert (CProc Buffer_get) (deleteMany [CIn Consumer_pull,(
        ↪ selectTimer 2 ci)] ci),pv)
  | otherwise = s
```

Listing 4.47: Example of awake function – Producer-Consumer problem

4.5.10 isWithinContextOf

The `isWithinContextOf` function is a helper function to test if the procedure of an agent is executed by a specific context agent (possibly indirectly). This is a helper function used by `fire` function when one of the transitions of `TDelay` 4.4.1, `TIn` 4.4.4, `TOut` 4.4.12, `STDelayEnd` 4.4.22, `STInPP` 4.4.19, `STLoopEnd` 4.4.24, `STInEnd` 4.4.23, `STOutEnd` 4.4.25, or `STOutPP` 4.4.21 is considered.

Actually, the `isWithinContextOf` is a family of functions that differ by the name and type of the required arguments. The instance of the `isWithinContextOf` function is generated only for active agents. For each active agent, the name of the specific version of the function is appended with the name of the agent. The first argument is the name of the procedure to be tested if is executed directly or indirectly by the active agent mentioned in the name of the function. The second argument represents the remote port that is owned by another agent and with which the communication is supposed to be established. The result of the function is the boolean value of **`True`** or **`False`**.

The algorithm generating the `isWithinContextOf` function was presented in Algorithm 4.36. For every active agent in the system that calls at least one procedure, the algorithm explores all the calling chains leading to any procedure (so-called **calling paths**). The calling path is a sequence of the calls originated by the **`in`** or **`out`** instruction of the active agent and calling a procedure of a passive agent, that possibly, calls another procedure of another passive agent and so on. All the calling paths are found in linear time complexity by a modified version of the depth-first search algorithm, which takes advantage of a full communication graph data structure 4.6. For each calling path originating from the given active agent, the algorithm chains conditions that check if the value of `CProc` representing the called procedure is on the direct caller-agent's context list.

The example of generated `isWithinContextOf` function for the Producer-Consumer problem is presented in Listing 4.48.

```

for each active agent that calls at least one procedure then
  for each procedure that can be executed within the context of the active agent then
    print "isWithinContextOf_{activeAgentName} {procedureName} {systemStatePatternMatching}"

      for each called procedure on the calling path then
        if callingPathCondition already exists then
          callerIndex ← index of the previous agent on that path
          callingPathCondition ← "{callingPathCondition} && member (CProc {calledProcedureName}) ci{callerIndex}"
        else
          callingPathCondition ← "member (CProc {calledProcedureName}) ci{activeAgentIndex}"
        end
      end
      print " | {callingPathCondition} = True"
    end
    print " | otherwise = False"
end
```

Algorithm 4.36: Algorithm generating isWithinContextOf helper function

```
isWithinContextOf_Consumer Buffer_get s@(state1@(am1,pc1,ci1,pv1@(pv1_1)),state2@(am2,pc2,ci2,pv2@(pv2_1, pv2_2)),state3@(
    ↪ am3,pc3,ci3,pv3@(pv3_1)))
  | member (CProc Buffer_get) ci3 = True
  | otherwise = False

isWithinContextOf_Producer Buffer_put s@(state1@(am1,pc1,ci1,pv1@(pv1_1)),state2@(am2,pc2,ci2,pv2@(pv2_1, pv2_2)),state3@(am3
    ↪ ,pc3,ci3,pv3@(pv3_1)))
  | member (CProc Buffer_put) ci1 = True
  | otherwise = False
```

Listing 4.48: Example of isWithinContextOf function – Producer-Consumer problem

4.6 Full communication graph

The full communication graph data structure extends the idea of the communication diagram 3.1. The main purpose of the full communication graph is to support fast searching of calling paths in the system. A calling path is a sequence of agents such that the previous agent in the sequence calls the next agent in the sequence. An active agent calling another active agent is a trivial case, as the calling path ends having just two agents. As another active agent cannot call third agent in that same instruction. However, there is an additional non-trivial case. When an active agent calls a passive agent's procedure, that passive agent may call another passive agent's procedure, which may result with a calling path longer than two agents. The later case poses a problem of efficient search of calling paths starting from an active agent and possibly going through several passive agent's procedures. As, the current compiler approach is to statically create predicates that select a specific calling path. Depending solely on the communication graph, it is not possible to find out if a procedure calls another passive agent's procedure, as there are two different ports of the passive agent involved. The communication graph does not contain the link representing a connection between two ports of the same passive agent. Those missing links are so-called **internal links**. Therefore, an internal link always leads from a procedural port of a passive agent to its non-procedural port if the communication instruction using the non-procedural port is executable from the procedure. Therefore, the full communication graph contains all edges and nodes from a communication diagram and additionally, it contains edges that represent internal connections within passive agents. The internal link between a procedural port and a non-procedural port of the same passive agent is added when the code of the procedure related to the procedural port contains communication instruction of `in` or `out` that communicates by non-procedural port. Once the full communication graph is constructed, finding all passive agents' procedures that can be executed directly or indirectly within the context of an active agent is reduced to the depth-first search algorithm on the full communication graph.

Formally, the full communication graph is defined as a tuple $G_{full} = (\boldsymbol{D}, \mathcal{C}_{full})$, where:

- $\boldsymbol{D}$ is a non-hierarchical Alvis model already discussed in the by the definition 3.1.1.
- $\mathcal{C}_{full} \subseteq (\mathcal{P} \times \mathbb{N}) \times (\mathcal{P} \times \mathbb{N})$ is full communication relation that is restricted by the following conditions:
  - The full communication relation is a union of two relations: $\mathcal{C}_{inter}$ – inter-connections between different agents, and $\mathcal{C}_{internal}$ – internal connection within passive agents.

$$\mathcal{C}_{full} = \mathcal{C}_{inter} \cup \mathcal{C}_{internal} \tag{4.1}$$

  - There is no connection that would be both an internal and an inter-connection.

$$\mathcal{C}_{inter} \cap \mathcal{C}_{internal} = \varnothing \tag{4.2}$$

  - Projection of the inter-connection relation on the Cartesian product of the two sets of ports of $\boldsymbol{D}$ (the Alvis model cf. Sec. 3.4) has to be equal to $\mathcal{C}$ (the communication relation of the communication diagram cf. Sec. 3.1).

$$\pi_{\mathcal{P}\times\mathcal{P}}(\mathcal{C}_{inter}) = \mathcal{C} \tag{4.3}$$

  - If there is a relation between (p, n) and (q, m) and ports p is the port of X and q is the port of Y (X is different than Y), it means that there has to be instruction $in\ p \mid T$ within code block of agent X at the program counter value of n and the corresponding instruction $out\ q \mid T$ within code block of agent Y at the program counter value of m or the other way round.

$$\begin{aligned}\forall_{X,Y\in\mathcal{A}}: & X \neq Y \wedge p \in \mathcal{P}(X) \wedge q \in \mathcal{P}(Y) \wedge ((p,n),(q,m)) \in \mathcal{C}_{inter} \implies \\ & \implies B_n(X) = in\ p\,|\,T \wedge B_m(Y) = out\ q\,|\,T \vee \\ & \vee B_n(X) = out\ p\,|\,T \wedge B_m(Y) = in\ q\,|\,T\end{aligned} \tag{4.4}$$

- All internal connections cannot link ports of different agents in the system.

$$\forall_{X,Y\in\mathcal{A}}: X \neq Y \implies (\mathcal{P}(X) \times \mathcal{P}(Y)) \cap \mathcal{C}_{internal} = \varnothing \tag{4.5}$$

- All internal connections cannot link ports of the same active agent.

$$\forall_{X\in\mathcal{A}_A}: (\mathcal{P}(X) \times \mathcal{P}(X)) \cap \mathcal{C}_{internal} = \varnothing \tag{4.6}$$

- If there is relation between (p,n) and (q,m) and ports p is the procedural port of Q and q is a non-procedural port of Q, it means that there has to be a procedure p within code block of agent Q and at the program counter value of n there is an instruction reading or sending data through procedural port $in\ p|T$ or $out\ p|T$ and there is a communication instruction through its non-procedural port q: $in\ q\,|\,T$ or $out\ q\,|\,T$ within code block of procedure $Q.p$ at the program counter value of m or the other way round.

$$\begin{aligned}\forall_{Q\in\mathcal{A}_P}: & p \in \mathcal{P}(Q) \wedge q \in \mathcal{P}(Q) \wedge ((p,n),(q,m)) \in \mathcal{C}_{internal} \implies \\ & \implies p \in \mathcal{P}_{proc}(Q) \wedge q \notin \mathcal{P}_{proc}(Q) \\ & \wedge (B_n(Q.p) = in\ p\,|\,T \vee B_n(Q.p) = out\ p\,|\,T\,) \\ & \wedge (B_m(Q.p) = in\ q\,|\,T \vee B_m(Q.p) = out\ q\,|\,T\,) \\ & \vee (q \in \mathcal{P}_{proc}(Q) \wedge p \notin \mathcal{P}_{proc}(Q)) \\ & \wedge (B_n(Q.p) = in\ p\,|\,T \vee B_n(Q.p) = out\ p\,|\,T\,) \\ & \wedge (B_m(Q.p) = in\ q\,|\,T \vee B_m(Q.p) = out\ q\,|\,T\,)\end{aligned} \tag{4.7}$$

Chapter 5

LTS generation algorithm

5.1 Algorithm design

In the previous chapter 4, we have discussed all the elements of the implicit representation of the state space of an Alvis formal model. The key part of the representation are definitions of functions `enable` and `fire`. Those functions are necessary to build an explicit representation of the LTS graph. The elements of the implicit representation of the state space were accompanied by algorithms that were implemented in the Alvis Compiler as part of this dissertation. Alvis Compiler allows creation of Intermediate Haskell Representation (IHR). IHR is a small domain-specific program generated for the given Alvis model. The code of the program describes the implicit LTS form. IHR program execution result is the explicit state space graph (LTS) which could be generated in various formats. The intermediate representation allows addition of support for different state space representation formats and implementation of on-the-fly model checking algorithms without modification of the compiler itself. The LTS graph can be further model checked with third party model checkers. There is, as well, a minimal on-the-fly model checker limited to invariants which can be run as part of IHR runtime – so called filtering functions.

In this chapter, the last missing part of the journey from model design to the explicit form of the state space graph in the form of LTS is outlined. The generic algorithm that allows to compute an explicit form of the graph leveraging IHR's `enable`, `fire`, and `s0` functions.

The process of building an explicit LTS graph poses many problems due to the state space explosion problem [180, 114]. Therefore, any optimal, even linear algorithm in terms of the number of states of the system is inevitably to fail at some point because of the exponential character of the input data. By failure, it is understood that the algorithm faces out of memory errors or the computation time is unacceptably long. By exponential complexity, we mean that the number of reachable states of the system grow exponentially with the number of lines of the code of the designed system. The factors that mostly affect the number of states is:

- number of active agent in the modelled system;
- local variables that can potentially have many internal states (like e.g. 32-bit integer variable has 4 billion different values and therefore 4 billion different internal states);

There are various methods to limit the number of states in the explored state space that were already mentioned in Sec. 1.8. The problem of limiting the number of states of the model is independent of the problem of exploration of the state space because even though the state space is constrained, and there are fewer states to be explored, there is still a necessity to develop an algorithm for exploration of constrained state space. However, just to put the discussed possible solution into Alvis context:

abstraction – the designer could change the model of the system into simpler version and therefore limit number of states. Alvis allows introducing arbitrary complicated computations in its **`exec`** command which is related to a single transition. Operations on several variables could be grouped into a single `exec` expression if the variables were grouped into a tuple of variables. Integer variables could be replaced with enumeration values with limited number of states. Finally, the model could be simplified with abstraction and removing details, reducing number of variables or number of agents, but that could hide the errors in the design.

reducing symmetries – Taking advantage of the potential symmetries in the state space [41, 89] may significantly reduce the number of states required to be explored. Alvis allows specifying multiple instance of the agent with the same behaviour, therefore the state space graph of the model will be symmetrical and swapping agent inside the tuple representing system state does not change final LTS. Therefore, an equivalence relation could be introduced that makes the states of the corresponding agents equivalent to each other.

As the previous approach needs to be addressed by the designer rather than resolved automatically from the tool perspective, we will limit our discussion in this chapter to the first two alternatives. Therefore, the problem of exploration of the implicit LTS graph as specified in the IHR and creation of its explicit form is discussed in the following section 5.2. The approach of limiting the state space before its exploration is discussed in Sec. 5.3.

Therefore, we will focus on the efficiency of the algorithm that explores the state space and creates an explicit LTS. One way of addressing the problem is by selection of the optimal algorithm for graph exploration. An optimal algorithm needs to limit its time complexity and memory complexity. The algorithm depends on the number of states n_s and the number of transitions n_t in the LTS. The LTS is a sparse graph as the number of outgoing transitions is proportional to the number of agents, which is constant for the given model. Therefore, one can assume that the number of transitions is proportional to the number of states: $n_t \propto n_s \propto n$, and further analysis can be simplified with the use of just n. The time complexity cannot be worse than $\mathcal{O}(n \log n)$ as n – the number of states is already problematic due to state explosion problem. Any algorithm with quadratic time complexity: $\mathcal{O}(n^2)$ cannot be accepted, as processing of 10^9 states would require ~ 10^{18} operations. That translates to years of execution when processing speed is of the order of 10^9 operations per second. While algorithm of complexity $\mathcal{O}(n \log n)$ would require ~ 10^{10} operations and therefore require seconds or minutes of computations.

Additionally, to the computational complexity, the algorithm should have a memory complexity of $\mathcal{O}(n)$. Moreover, even then we have to consider the memory footprint of the algorithm, considering locality of the data and the latency, throughput of the data access and the memory storage capacity of the various data storage systems (main memory, hard drive (SSD vs HDD), internet connection to external system) in the computation system [71]. The main memory resources of the contemporary computers that are fast to access can be depleted with a matter of several minutes of computations. Therefore, to be able to calculate bigger systems, an attention has to be paid to the storage of the data.

5.2 Primary algorithm

In terms of an Alvis formal model, the LTS is a graph, where vertices are system states, and labelled edges are transitions of the system. So, the problem of generation of the explicit LTS can be reduced to the problem of graph traversal. The problem of the **graph traversal** with the goal of visiting every vertex of the graph and its every edge only once is well-known in the literature [158, 48]. There are two primary algorithms of graph traversal, namely: depth-first search (**DFS**) and breadth-first search (**BFS**). The only difference between those two algorithms is the order in which they visit the vertices of the graph. While vertices of the graph are explored, they go through three different states when the algorithm performs the search. Those states of the vertex are: *undiscovered*, *discovered*, *processed*. Both of the algorithms start with all the vertices in the *undiscovered* state. The algorithms take advantage of the container of already discovered yet not processed states. An algorithm then processes states from the container one by one. The algorithms operate as follows: a vertex from the container is taken, and all states reachable from it are calculated. Then all those reachable states are processed. The processing of a vertex tests its state and if it is *undiscovered* then its state is changed to *discovered*, and the reachable vertex is placed into the container. Otherwise, the reachable vertex is omitted (because it is already processed or waits in the container for its turn). When all outgoing edges are explored, the source vertex state is marked as *processed*. An algorithm proceeds by taking another vertex from the container and repeating the steps until no vertices are left in the container. The only distinction between DFS and BFS is the data structure used as the container:

- *Stack* - when the discovered vertices are stored into the last-in, first-out (LIFO) stack. Then the algorithm is the implementation of the **DFS**.
- *Queue* - when the discovered vertices are stored into the first-in, first-out (FIFO) queue. Then the algorithm is the implementation of the **BFS**.

The following sections will discuss the actual implementation in much further details. Some optimization will be discussed together with actual implementation. The algorithm implementation leverages extensive usage of Haskell classes. As a result, the particular data structure implementations can be easily substituted and its relative efficiency can be compared. Making the algorithm to depend on classes rather than concrete instances is realization of the dependency inversion principle [123, 124]. The dependency inversion principle allows then injection of any data structure that conforms to the Haskell class required by the algorithm and allows testing various implementations easily on real data without necessity to modify the proper algorithm.

5.2.1 LTS primary computation loop

This section will discuss the primary computation loop – the heart of the graph traversal algorithm. The data structures that are dependencies of this algorithm are discussed in detail in later sections. Here, only short description will be provided. The main difference between the implementation presented in Listing 5.1 and the general and classical description of graph traversal problem provided in Sec. 5.2 is that there is no explicit set that would keep track of the state of the vertices (*undiscovered*, *discovered*, *processed*). This is because the set of vertices is unknown a priori and the main goal of the algorithm is to find that set. The track of the state of the vertices is maintained implicitly by the data structures that are used to track discovered vertices. Thanks to that, the algorithm is more easily scalable, and it requires less memory.

The container implementation where the discovered states are stored temporarily is provided as an instance of Haskell class `ToVisitService toVisitService`, cf line 2 and Sec. 5.2.4. The class provides the method `nextUnvisitedStateMaybe` to request next unprocessed vertex, cf line 14. The value returned by the method is wrapped into the data container **`Maybe`** that can hold up to one value. When a value **`Nothing`** of data type **`Maybe`** is returned, then it signifies that the container has run out of states to be explored. Otherwise, the state to be processed is returned wrapped into a single value definition of **`Just`** `value` of data type **`Maybe`**.

The Haskell class `ExploredService exploredService`, cf. line 4, abstracts away the implementation of the structure storing all discovered and processed states. The class will be discussed in detail in Sec. 5.2.7.

The Haskell class `GraphDescription model`, cf. line 7, abstracts away implementation of the particular Alvis model, i.e. its implicit representation as an IHR generated by the Alvis compiler. The class will be discussed in detail in Sec. 5.2.6.

The Haskell class `Hasher hasher`, cf. line 3, abstracts away the implementation of the method to calculate hash of the state. Hashing is a way to optimize the memory footprint of the algorithm by compacting the vertex of the graph. Therefore, it could be achieved significant reduction of the amount of data stored in main memory. The class will be discussed in detail in Sec. 5.2.5.

The Haskell classes of `ExploredService exploredService`, `GraphDescription model` ↪ , and `ToVisitService toVisitService` are parameterized with the type of the vertex in the graph for the specific model, cf. line 5 and type of the graph edge for the specific model, cf. line 6.

The last dependency that was not yet described is the `NextIdGenerator`, cf. line 11. The generator is in charge of providing consequent numbers that tag the state uniquely.

The function `ltsComputationLoop` retrieves the system state from the container of unprocessed states, then it checks whether the returned state is actually present or stops the computation if the container returned **`Nothing`**, cf line 15. The method `while` is one of the method from the monad transformer `LoopWhileT` which allows processing computation in do–while loop. The next step may happen only if the condition evaluates to true by function `while`, then a retrieved state is passed for further processing to the function `proccessState`, cf. line 17 and Sec. 5.2.2 Apart from the `nextState` all the data structures are passed as arguments that describe context of the computations. The method `lift` is a standard function from the class `MonadTrans` and allows embedding computation of one monad into another one. Here, **`IO`** results in `LoopWhileT`.

```
1  ltsComputationLoopIO :: (
2    ToVisitService toVisitService,
3    Hasher hasher,
4    ExploredService exploredService,
5    State stateType,
6    TTransition transitionType,
7    GraphDescription model) =>
8      toVisitService stateType transitionType ->
9      hasher ->
10     exploredService stateType transitionType ->
11     NextIdGenerator ->
12     model stateType transitionType -> LoopWhileT (IO) ()
13 ltsComputationLoopIO toVisitService hasher exploredService idGenerator model = do
14   mNextState <- lift (nextUnvisitedStateMaybe toVisitService)
15   while (isJust mNextState)
16   let nextState = fromJust mNextState
17   lift (processState toVisitService hasher exploredService idGenerator model nextState)
```

Listing 5.1: ltsComputationLoopIO

5.2.2 State processing method

This section discusses the processing of a single vertex of the graph that represents a single state of the system of the LTS. While processing a system state, all enabled transitions in that state are computed together with all reachable states. At this stage of computation, the state of the vertex is *discovered* but not yet processed. After execution of the methods `processState` together with `visitTransition`, the state of that vertex changes to *processed*, and it will not be explored any further, cf. Listings 5.2 and 5.3.

The definition of the function `processState` is shown in Listing 5.2. Its parameters were already outlined in Sec. 5.2.1 together with `ltsComputationLoopIO` function:

- `ToVisitService toVisitService`,
- `Hasher hasher`,
- `ExploredService exploredService`,
- `GraphDescription model`,
- `NextIdGenerator`.

The only parameter that was not previously encountered is the `QueuedState stateType` argument, cf. line 13. That argument represents a vertex and its metadata to be processed by the function. The argument is a 2-tuple that consist of the vertex (system state) and its associated unique identifier. The state was retrieved from the queue of unprocessed states and now is being processed by this function.

In the first step of the state processing method, cf. line 15, calculation of all outgoing edges from the given vertex (`state`) takes place. To achieve this goal one of the methods from `GraphDescription` ↪ `model` is called, namely `enableTransitions`, cf. Section 4.3. As `GraphDescription` represents any Alvis model in generic way.

The second and the last action of the method, cf. line 16, is the call of the `visitTransition` for every enabled transition (outgoing edge) calculated in the previous step (`transitionsToExplore`) and the current system state (current vertex). The iteration over all `transitionsToExplore` is done by a built-in function **`mapM_`** which allows executing IO action for every element of the given list.

```
1  processState :: (
2    ToVisitService toVisitService,
3    Hasher hasher,
4    ExploredService exploredService,
5    State stateType,
6    TTransition transitionType,
7    GraphDescription model) =>
8      toVisitService stateType transitionType ->
9      hasher ->
10     exploredService stateType transitionType ->
11     NextIdGenerator ->
12     model stateType transitionType ->
13     QueuedState stateType -> IO ()
14 processState toVisitService hasher exploredService idGenerator model (id, state) = do
15   let transitionsToExplore = enableTransitions model state
16   mapM_ (visitTransition toVisitService hasher exploredService idGenerator model (id, state)) transitionsToExplore
```

Listing 5.2: processState

The definition of the function `visitTransition` is shown in Listing 5.3. All parameters of the function `visitTransition` were previously described. The function calculates the next vertex of the LTS graph (state of the system) that is directly reachable from the given vertex by the given edge (transition). The function is as well responsible for updating the context data structures like, e.g., `ExploredService`.

The first action of the method calculates all reachable states from the given vertex `fromState` through the given edge `transition`. The method `fire` belongs to the Haskell class `GraphDescription` ↪ and an instance of the classes delegates it invocation to the particular method `fire` for the model, cf. Section 4.4. In every case, there is a single state reachable by the particular transition, but in the case of the function `pick`, the transition is not deterministic and there are more than one possible reachable states. In terms of the generated graph, for every such a case, the dedicated edge is generated, so this is still an ordinary graph.

The second action, cf. line 17, computes hash for every vertex returned by the former action. Any implementation of the hash function may be provided as long as the `hasher` belongs to the Haskell class `Hasher` and, therefore, implements its method `stateHash`. More detailed discussion of the `Hasher` is presented in Sec. 5.2.5, but it is worth to mention here that its purpose is to provide compaction of the state to limit the amount of data stored in main memory.

The third action, cf. line 18, checks every generated vertex and checks if it was not already *processed* nor *discovered*. The instance of the Haskell class `ExploredService` is responsible for both storing already discovered states and allows testing whether the given state is still *undiscovered* or it was already seen. The following functionality is realized by the method `findAlreadyExploredStates`. The method returns a list of `State stateType =>Maybe (Int, stateType)`, which is an optional tuple of the vertex (system state) and its associated meta-data. When the vertex is in the *undiscovered* state, then the constructor **`Nothing`** is returned, otherwise, the constructor **`Just`** containing the tuple is returned.

The next two lines, cf. line 19 and 20, combine the vertex and its index with its previously calculated hash. Then they partition the list of values containing **`Nothing`** or **`Just`** from the previous action into two lists, where the first one (`alreadyStoredStates`) contains only the values already stored, and the second list (`toBeStoredStates`) contains vertices that are still *undiscovered*.

The sixth action, cf. line 21, extracts the index of the already *discovered* vertex and creates the tuple that represents the edge label data `transition` together with the index of the state from where the edge start `fromStateId` and the index of the state to where the edge leads. The later value comes from the list `alreadyStoredStates`, and therefore the result is again a list of edges.

The next action, cf. line 22, assigns the new indexes to the vertices in the state *undiscovered*. The indexes are generated by the `NextIdGenerator`.

The next action, cf. line 23, stores the *undiscovered* vertices in the instance of `ExploredService`

↪ with the method `addExploredStates`. So from this point, every vertex from the list `toBeStoredStates` changes its state form *undiscovered* to *discovered*.

The next action, cf. line 25, processes recently *discovered* vertices from the list `toBeStoredStates` ↪ and attaches the identifiers assigned to them by the implementation of the Haskell class `ExploredService` from the previous step.

In the next action, cf. line 26, the states prepared in the previous step are inserted into the container `ToVisitService toVisitService` with a call to the function `scheduleStatesToVisit`. This operation schedules states for future processing. `ToVisitService toVisitService` can be instantiated with stack or queue and therefore DFS or BFS algorithm can be chosen.

In the next action, cf. line 28, the index of the newly *discovered* vertices is extracted. Then for every such index, a 3-tuple that represents the edge label data `transition` is created. The tuple contains an index of the vertex from where the edge start `fromStateId` and the index of the state to where the edge leads. The later value comes from the list `ids`, and therefore the result is again a list of edges.

In the last action, cf. line 29, all transitions are stored by the method `addExploredTransitions` in the instance of the `ExploredService`.

The last section below the algorithm that starts after the keyword **`where`** contains a few helper functions. Its purpose is to increase the readability of the algorithm by adding a meaningful name to simple computations. The functions are one-liners that are responsible for transforming data structures one into another, combine several arguments into a tuple, or test the type whether it was created with the constructor **`Just`**.

```
1  visitTransition :: (
2    ToVisitService toVisitService,
3    Hasher hasher,
4    ExploredService exploredService,
5    State stateType,
6    TTransition transitionType,
7    GraphDescription model) =>
8      toVisitService stateType transitionType ->
9      hasher ->
10     exploredService stateType transitionType ->
11     NextIdGenerator ->
12     model stateType transitionType ->
13     QueuedState stateType ->
14     transitionType -> (IO) ()
15 visitTransition toVisitService hasher exploredService idGenerator model (fromStateId, fromState) transition = do
16     let newStates = fire model transition fromState
17     let hashedStates = map (\s -> (stateHash hasher s, s)) newStates
18     lookedUpStates <- findAlreadyExploredStates exploredService hashedStates
19     let zippedStates = zip hashedStates lookedUpStates
20     let (alreadyStoredStates,toBeStoredStates) = partition (isAlreadyStored) zippedStates
```

```
    let toBeStoredTransitions = map (prepareTransitionToStoreToAlreadyStoredState fromStateId transition) alreadyStoredStates
    preparedToBeStored <− mapM (addIndexToHashedState) toBeStoredStates
    ids <− addExploredStates exploredService preparedToBeStored

    let toBeScheduledStates = zipWith (prepareStateToStore) toBeStoredStates ids
    scheduleStatesToVisit toVisitService toBeScheduledStates

    let toBeStoredTransitions2 = map (prepareTransitionToStoreFromNewStoredIds fromStateId transition) ids
    addExploredTransitions exploredService (toBeStoredTransitions ++ toBeStoredTransitions2)

    return ()
    where
      addIndexToHashedState ((hash,state), _) = do
        id <− nextId idGenerator
        return (hash,(id,state))

      isAlreadyStored (_,lookedUpState) =
        isJust lookedUpState

      prepareTransitionToStoreToAlreadyStoredState fromStateId transition (_,Just (toStateId,_)) =
        (fromStateId,transition,toStateId)

      prepareTransitionToStoreFromNewStoredIds fromStateId transition toStateId =
        (fromStateId,transition,toStateId)

      prepareStateToStore ((hash,s),Nothing) toStateId =
        (toStateId, s)
```

Listing 5.3: visitTransition

5.2.3 Time complexity of ltsComputationLoop

In this section, the time complexity of the computation loop will be discussed. The code listings from Sec. 5.2.2 and 5.2.2 are crucial to the analysis. The function `processState` and `nextUnvisitedStateMaybe` are called once for every vertex in the graph, cf. Listing 5.1. That is because the states come from the container represented as an instance of `ToVisitService`, and therefore, those are only *unprocessed* vertices. The function `visitTransition` is again called once for every edge in the graph, cf. Listing 5.2. That is because even though the function `visitTransition` is called by the `processState` it is called only for the edges outgoing from the vertex being visited. So, the complexity of the algorithm is no smaller than $\mathcal{O}(V + E)$ where V is the number of vertices, and E is the number of edges. However, the LTS graph is very sparse, there are only up to several transitions outgoing from any vertex (number of transitions outgoing from the single vertex is proportional to the number of agents in the modelled system). Therefore, E is proportional to V, and therefore the number of times when the function `visitTransition` is called can be stated as $\mathcal{O}(V)$, as $E\ V$.

Another factor that has to be accounted comes from the methods called by the function `visitTransition` that are implemented in the supporting data structures. Therefore, from Listing 5.3 the below list of functions has to be analysed. All functions will be discussed in more detail in next sections. The complexity is measured with the number of states of the system, therefore as most of the functions depend solely on the current state, their complexity is constant.

- `fire` – $\mathcal{O}(1)$ complexity;
- `stateHash` – $\mathcal{O}(1)$ complexity;
- `findAlreadyExploredStates` – usually $\mathcal{O}(\log(n))$ or amortized $\mathcal{O}((1))$ complexity depending on the implementation;
- `addExploredStates` – it usually has $\mathcal{O}(\log(n))$ or amortized $\mathcal{O}((1))$ complexity depending on the implementation;
- `scheduleStatesToVisit` – amortized $\mathcal{O}(1)$ complexity;
- `addExploredTransitions` – usually $\mathcal{O}(\log(n))$ or amortized $\mathcal{O}((1))$ complexity;
- other local computations have $\mathcal{O}(1)$ complexity.

In summary, the effective time complexity of the LTS graph generation depends on the data structures used. In the most general case, however, the complexity is $\mathcal{O}(n\log(n))$, which was the goal in the first place. This is because the method `addExploredStates` is called $\mathcal{O}(n)$ times and its internal complexity can be $\mathcal{O}(\log(n))$.

5.2.4 ToVisitService

The Haskell class `ToVisitService`, cf. listing 5.4, abstracts away the necessary functionality of the container mentioned in Sec. 5.2. The minimal required functionality is the possibility to store vertices of the graph that are yet to be processed into the container. Additionally, any stored vertex has to be retrieved at some point. So the container cannot lose track of any vertex. If there are no more vertices in the container, then the special value is returned when one tries to retrieve a vertex. The method `scheduleStatesToVisit` provides the functionality to store the vertices into the container. The method `nextUnvisitedStateMaybe` provides the functionality to retrieve a single vertex from the container, retrieving the vertex removes it from the container. When the container holds some vertices, then `nextUnvisitedStateMaybe` is successful and the value constructor **`Just`** with the vertex is returned wrapped in the action **`IO`**. When the container is empty, the value constructor **`Nothing`** wrapped in the action **`IO`** is returned. The later result signifies that the container is already empty and there are no more vertices to visit, therefore exploration of the graph can be finished. Wrapping the result in **`IO`** monad allows providing implementation of the container with non-pure implementations like e.g. disk-based queue implementations.

```haskell
type QueuedState stateType = (StateId,stateType)

class ToVisitService a where
    nextUnvisitedStateMaybe :: (State stateType, TTransition transitionType) => a stateType transitionType -> IO (Maybe (QueuedState stateType))
    scheduleStatesToVisit :: (State stateType, TTransition transitionType) => a stateType transitionType -> [QueuedState stateType] -> IO ()
```

Listing 5.4: ToVisitService

In Listing 5.5, one of the possible instances of the class `ToVisitService` is presented. The type implements a FIFO queue that resides only in main memory. Therefore, if this implementation is used in the graph traversal algorithm, then the implemented graph traversal algorithm is **BFS**, as it was already discussed in Sec. 5.2. From the design pattern perspective, the implementation is an example of the adapter pattern [65]. The adapted method is from the Haskell library `Data.Sequence` [181]. Both of the required operations has amortized time complexity of $\mathcal{O}(1)$. Moreover, the library provides the best appending and retrieving-the-head operations from the tested structures in the benchmark [184]. The actual implementation is based on the 2-3 finger trees [77] which is a purely functional data structure that allows to implement random access element in amortized logarithmic time, but accessing, appending front and end of the container requires amortized constant time.

The implementation of the method `nextUnvisitedStateMaybe` calls the method `tryPoll` as it can be observed in the line 16. Eventually, the method `viewr` from the library `Data.Sequence` is called, which takes the element form the right-most end of the queue, and its result is wrapped into an instance of **`Maybe`**. Then, the reference `STRef` is updated with the modified queue.

The implementation of the method `scheduleStatesToVisit` calls the method `offer` for every provided vertex, `newStates` in a loop, as it can be observed in the line 20. Eventually, the operator `<|` from the library `Data.Sequence` is called, which adds the element to the left-most end of the queue, and the reference `STRef` is updated with the modified queue.

```haskell
1  type QueueST s a = STRef s (Seq.Seq a)
2  type Queue a = QueueST RealWorld a
3  type UnvisitedStatesST s stateType = QueueST s (QueuedState stateType)
4  type UnvisitedStates stateType = Queue (QueuedState stateType)
5
6  data InMemoryQueue stateType transitionType = InMemoryQueue (UnvisitedStates stateType)
7
8  mkInMemoryQueue ::(State stateType, TTransition transitionType) => IO (InMemoryQueue stateType transitionType)
9  mkInMemoryQueue = do
10   unvisitedStates <- emptyQueue
11   return (InMemoryQueue unvisitedStates)
12
```

```
instance ToVisitService InMemoryQueue where
  --nextUnvisitedStateMaybe :: InMemoryQueue -> IO (Maybe QueuedState)
  nextUnvisitedStateMaybe (InMemoryQueue queue) = do
    tryPoll queue

  --scheduleStatesToVisit :: InMemoryQueue -> [QueuedState] -> IO ()
  scheduleStatesToVisit (InMemoryQueue queue) newStates = do
    mapM_ (offer queue) newStates

emptyQueue :: (State stateType) => IO (UnvisitedStates stateType)
emptyQueue = stToIO emptyQueueST

offer :: Queue a -> a -> IO ()
offer q = stToIO . offerST q

tryPoll :: Queue a -> IO (Maybe a)
tryPoll = stToIO . tryPollST

emptyQueueST :: (State stateType) => ST s (UnvisitedStatesST s stateType)
emptyQueueST = newSTRef Seq.empty

offerST :: QueueST s a -> a -> ST s ()
offerST q e = do
    queue <- readSTRef q
    let newQueue = e <| queue
    writeSTRef q newQueue

tryPollST :: QueueST s a -> ST s (Maybe a)
tryPollST q = do
    queue <- readSTRef q
    let end = Seq.viewr queue
    process end
    where
      process Seq.EmptyR = return Nothing
      process (newQueue :> element) = do
        writeSTRef q newQueue
        return (Just element)
```

Listing 5.5: InMemoryQueue

5.2.5 Hasher

The Haskell class `Hasher` follows the single responsibility principle [125, 126] – it provides a single functionality of the hash of the system state computation, cf. Listing 5.6. Because the graph generation algorithm depends on the abstract class of Hasher, it allows substituting different hashing algorithm without modification of the graph traversal algorithm implementation. The hashing is not necessary from the graph exploration algorithm perspective, but the hash can be used by an instance of the class `ExploredService` to optimize the main memory usage or speed up the vertex look-up operation, cf. sec. 5.2.7. The instance of the class requires an implementation of a single function `stateHash` that performs hashing operation. The method accepts the instance of the `Hasher` and the vertex (a generic state of the system `(State ↪ stateType)=>stateType`). It returns the hash as a `ShortByteString` – one of the Haskell string implementation [161].

```haskell
type StateHash = ShortByteString
type HashedState stateType = (StateHash, stateType)

class Hasher hasher where
  stateHash :: (State stateType) => hasher -> stateType -> StateHash
```

Listing 5.6: Hasher

We can distinguish two groups of hashing algorithms that differ by their purpose [64]. One group consists of cryptographic hashes and another one of non-cryptographic hashes. The former group is used in cryptography and therefore a hashing function is required to be:

- non-reversible – it should not be possible to calculate original message having its hash.
- deterministic – the same message should always generate the same hash.
- collision resistant – it should be rare that two different messages have the same hash.
- diffusive – almost identical messages that differ even by a single character should have non-similar hashes.
- slow to compute – (for some hashing function used in hashing passwords) it should be slow to compute so when an attacker gets access to the hashes, he will not be able to guess the original message by brute-force generation of all the possible hashes (as it should take years).

The later group is used in various data structures algorithm (hash maps, bloom filters) to speed up the search of the elements. Therefore, a hashing function has different requirements:

- deterministic – it still has to be deterministic, so one can replace comparison of the original data with the comparison of the hashes.
- fast to compute – in contrary to cryptographic hashes, computation of the hash should not waste CPU cycles.
- collision resistant – it should be relatively rare that two different messages have the same hash, but it is less strict than in the case of cryptographic hashes. Cryptographic hashes are used as a one-way function and only hash is stored in the system. Therefore, if there is necessity to compare if the message is the same as original one, the hash of that message is computed and compare against the stored hash. When the collision of the hashes happens, the algorithm will assume that the original message was the same. Therefore, collision for cryptographic hashes are extremely dangerous. For

non-cryptographic hashes, the original message is usually known, and the hash collisions can be resolved by other means. Therefore, collisions for non-cryptographic hashes can downgrade the look-up speed, but are not dangerous.

Some examples of cryptographic hash functions:

- MD5 – a cryptographic hash function with hash length of 128 bits.
- SHA-1 – a cryptographic hash function with hash length of 160 bits.
- SHA-256 [148] – a cryptographic hash function with hash length of 256 bits.
- bcrypt – a cryptographic hash function designed for slow computations and use in passwords encoding.

Some examples of non-cryptographic hash functions:

- CRC32 [50] – a popular checksum algorithm which produces a 32-bit number. What is worth to mentioned there is hardware support in modern Intel processor for that computation.
- Fowler-Noll-Vo (FNV1 and FNV1a) – family of hashes with hash lengths between 32 up to 1024 bits.
- Murmur3 [81, 7] – hash functions that produce hashes with 32 and 64 bit lengths.
- Spooky V2 [90] – family of hashes with hash lengths between 32 up to 128 bits.
- CityHash [151] – family of hashes created by Google with hash lengths between 32 up to 256 bits. They may leverage hardware supported CRC32 instruction.
- xxHash [46, 120] – family of hashes with hash lengths between 32 up to 128 bits.
- FarmHash [150, 72] – family of hashes created by Google with hash lengths between 32 up to 128 bits.

The comparison of the different hashes can be found in the benchmark [8, 178] – a test suite which evaluates collision, diffusiveness and randomness qualities of hash functions.

From the graph exploration algorithm perspective, the hasher is required to be: deterministic, collision resistant, and fast-to-compute. Collision resistance increases with the length of the hash. The probability that there is no collision between any two hashes when there was r values hashed and the hash length is n-bits is:

$$p = \frac{(2^n)!}{2^{nr}(2^n - r)!} \approx exp\left(\frac{-r(r-1)}{2^{(n+1)}}\right) \tag{5.1}$$

That probability decreases fast with the number of elements hashed r which is known as so-called birthday paradox and traditionally assigned to Richard Mises [183]. In table 5.1 probability of collision $1 - p$ is provided for three different hash lengths: 32, 64, and 128, and four different sizes of the state space: 10^4, 10^6,10^8, 10^9. As a reference to the given probability, it is worth to mention that probability of an error that different value is read from the value written to the solid-state storage: UBER (unrecoverable bit error rate) is between 10^{-18} and 10^{-15}. Therefore, 128-bit length of the hash should be enough for problems trackable in modern computers, while 32 and 64-bit hashes are not enough.

Table 5.1: Probability of a collision for hash length n and the size of the state space r

n \ r	10^9	10^8	10^6	10^4
32	1	1	1	1.16e-3
64	2.68e-2	2.72e-4	2.72e-8	2.72e-12
128	1.5e-21	1.5e-23	1.5e-27	1.5e-31

Several instances of the class `Hasher` were implemented and tested. Below, two implementations will be discussed with code examples. In Listing 5.7, one of the possible instances of the class `Hasher` is presented. The implementation uses SHA-256 cryptographic algorithm [148]. From the design pattern perspective, the implementation is an example of the adapter pattern [65]. The hash implementation provides a good trade-off between speed and lack of collisions. As a cryptographic hash, it is supposed to be relatively slower. However, the Haskell implementation is a wrapper for well established and highly optimised library written in pure C. The implementation of the method `stateHash` calls the method `hash`, cf. line 7 from the Haskell library `Crypto.Hash.SHA256` [74]. The library method, however, requires the string representation of the vertex. Therefore, before the call, the binary encoding of the vertex is calculated by the library `Data.Store` [159], cf line 9. The implementation that uses other hashing functions provided by different libraries is similar to the presented one. The hash method return binary string that consist of 32 bytes therefore only first 16 bytes are taken. The additional conversion from ByteString into ShortByteString is required due to the bug in the ByteString library discussed in the next chapter.

```
1  data SHA256Hasher = SHA256Hasher
2
3  mkSHA256Hasher :: IO SHA256Hasher
4  mkSHA256Hasher = return SHA256Hasher
5
6  instance Hasher SHA256Hasher where
7    stateHash SHA256Hasher s = (toShort . take 16 . SHA256.hash . toByteString) s
8      where
9        toByteString = Store.encode
```

Listing 5.7: SHA256Hasher

In Listing 5.8, another possible instance of the class `Hasher` is presented. The implementation uses a custom method that is optimised for the model prepared for testing of the LTS generation algorithm 6.2. The optimisations are based on the following observations:

- The mode of the agents does not change in that specific model (agents are always in the running mode).
- The context list of any agent in the model is always `empty`.

- The program counter value of any agent is always 1 or 2.
- There are two variables for every agent. Let us assume, that first variable x is changing within the range 0 and 63. The second variable y is constant.

Therefore:

- The mode of the agent and value of its context list can be neglected as those are constant.
- Only first two bits of the program counter are necessary, and therefore all program counter values can be packed within a single byte.
- Only first six bits of the first variable are necessary, and therefore all values can be packed within just three bytes.
- The constant variable can be neglected.

The implementation of the above approach is presented in the line 12. Variables $pc1 \ldots pc4$ represent program counter values of first, ..., and forth agent respectively. Variables $v1 \ldots v4$ represent values of the variable x of first, ..., and forth agent respectively. In order to allow bitwise operations, Haskell provides the following bit level operators: `.|.` – bitwise or, `.&.` – bitwise and, `` `shiftL` `` – bitwise shift left, and `` `shiftR` `` – bitwise shift right with sign. Then, all values of program counters are packed into the first byte such that the two least significant bits of $pc1$ are packed into the two least significant bits of the result, the two least significant bits of $pc2$ are packed into next two bits, and so on. The fifth and fourth bits of every variable v are packed into the second byte in the similar way as pc variables. The 4 least significant bits of first and second agent's variable x are packed into the third byte, and lastly, the 4 least significant bits of third and fourth agent's variable x are packed into the fourth byte.

```
1  instance Hasher CustomTestModelHasher where
2     stateHash CustomTestModelHasher = BS.pack . toCustomList
3
4  toCustomList :: (State stateType) => stateType –> [Word8]
5  toCustomList s = toCustomTestModelList ((unsafeCoerce s)::M10KState)
6
7  toCustomTestModelList :: M1048KState –> [Word8]
8  toCustomTestModelList ((_, pc1, _, (v1, _)),(_, pc2, _, (v2, _)),(_, pc3, _, (v3, _)),(_, pc4, _, (v4, _))) =
9     map fromIntegral [(pc1 .&. 0x03) .|. ((pc2 .&. 0x03) ‘shiftL‘ 2) .|. ((pc3 .&. 0x03) ‘shiftL‘ 4) .|. ((pc4 .&. 0x03) ‘shiftL‘ 6),
10 ((v1 .&. 0x30) ‘shiftR‘ 4) .|. ((v2 .&. 0x30) ‘shiftR‘ 2) .|. ((v3 .&. 0x30)) .|. ((v4 .&. 0x30) ‘shiftL‘ 2),
11 (v1 .&. 0x0F) .|. ((v2 .&. 0x0F) ‘shiftL‘ 4),
12 (v3 .&. 0x0F) .|. ((v4 .&. 0x0F) ‘shiftL‘ 4)]
```

Listing 5.8: Custom Hasher for test models

It is possible to represent values of the variable x from range 0 up to 127 in just 4 bytes as well. By taking advantange of the fact that pc values have only two distinct values and could be stored within just 1 bit. Then additional sixth bit of the v variable could be stored within 4 bits of the first byte of the result. However, storing values exceeding 127 requires more bytes of the result.

The implementation based on sha 256 or other hashing function requires more space due to its random output and birthday paradox but they are suitable for arbitrary Alvis model. On the other hand custom hasher is highly optimised and requires quater of the space of general purpose hashing but it is not suitable for any other Alvis model. For every model one would need to provide a custom implementation for that specific model.

5.2.6 GenericModel

The Haskell class `GenericModel` abstracts away the specific Alvis model implementation in the IHR, cf. listing 5.9. It makes the LTS computation algorithm independent of the specific model, and therefore, several Alvis models can be tested and benchmarked within the single application.

The class is presented in Listing 5.9 together with additional classes:

- `State` - is the generalization of the system state (vertex of the LTS graph);
- `TTransition` - is the generalization of the system transition (edge of the LTS graph);
- `DOTFormattable` - is the class covering the formatting of its instances in the DOT format (the DOT format is suitable for LTS visualization).

There are three classes that the class `State` extends, namely `Store`, `ToJSON`, and `FromJSON`. Those classes are required for serialisation of the states of the system (or its transitions). The class `Store` comes from the Haskell library `Data.Store` [159] and is responsible for efficient serialization and deserialization in the binary format supported by the library. The classes `ToJSON` and `FromJSON` come from the Haskell library `Data.Aeson` [141] and are responsible respectively for serialisation and deserialisation in the JSON format. Json format is a human-readable format but is less efficient in terms of memory required than the format provided by the library `Store`. Finally, the class `GenericModel` generalizes the Alvis model's IHR. When an arbitrary Alvis model provides instances of the above classes in the application code, its LTS model can be calculated alongside other Alvis models. The class `GenericModel` contains three methods:

- `enableTransitions` – It calculates all the edges outgoing from the given vertex, cf. line 9 and section 4.3. The method requires the vertex `systemType` and the model of the system parameterized by the types of its vertices and edges `model systemType transitionType`. The function returns the list of transitions `[transitionType]`
- `fire` – It calculates the vertex which is the successor of the given vertex and it is directly reachable by the given edge, cf. line 10 and section 4.4. The method requires the vertex `systemType`, the edge `transitionType` and the model of the system parameterized by the types of its vertices and edges `model systemType transitionType`. The function returns a list of vertices `[systemType ↪ ]`.

- `s0` – It calculates the initial state of the system, cf. line 11 and section 4.5.7. The method requires the system parameterized by the types of its vertices and edges `model systemType` ↪ `transitionType`. The function returns the single vertex that is the initial state of the modeled system `systemType`.

```
1  class (DOTFormattable stateType, Store stateType, ToJSON stateType, FromJSON stateType) => State stateType
2
3  class (DOTFormattable transitionType, Store transitionType, ToJSON transitionType, FromJSON transitionType) => TTransition
       ↪ transitionType
4
5  class Show a => DOTFormattable a where
6      toDOT :: a -> String
7
8  class GraphDescription model where
9      enableTransitions :: (TTransition transitionType, State systemType) => model systemType transitionType -> systemType -> [
           ↪ transitionType]
10     fire :: (TTransition transitionType, State systemType) => model systemType transitionType -> transitionType -> systemType -> [
           ↪ systemType]
11     s0 :: (TTransition transitionType, State systemType) => model systemType transitionType -> systemType
```

Listing 5.9: GenericModel

An example instance of an arbitrary Alvis model IHR `AlvisIHRModel` that implements the instance of the above class `GraphDescription` is presentented in Listing 5.10. All symbols that belong to the module `AlvisIHRModel` are addressed as `A.symbol` because of the qualified import in the line 1. In the lines from 3 up to 8 the definition of the data supporting model is made and aliases of the type representing the vertex `AState` and edge `ATransition` are created. The new type is necessary for the instance declaration in the lines from 10 up to 13. In the end of the listing are listed instances necessary to provide serialization and deserialization of the vertices and edges of the Alvis model starting from the line 15. The code is automatically generated by the Generic Haskell extension therefore instances are empty.

```
1  import qualified AlvisIHRModel as A
2
3  mkAModel :: IO AModel
4  mkAModel = return ADescription
5  data ADescription stateType transitionType = ADescription
6  type AState = A.State
7  type ATransition = A.TTransition
8  type AModel = ADescription AState ATransition
9
10 instance GraphDescription ADescription where
11     enableTransitions ADescription state = unsafeCoerce (A.enableTransitions ((unsafeCoerce state)::A.State))
12     fire ADescription transition state = unsafeCoerce (A.fire ((unsafeCoerce transition)::A.TTransition) ((unsafeCoerce state)::A.State))
13     s0 ADescription = unsafeCoerce A.s0
14
15 instance ToJSON A.Mode where
```

```
    toEncoding = genericToEncoding defaultOptions
instance FromJSON A.Mode
instance ToJSON A.TTransition where
    toEncoding = genericToEncoding defaultOptions
instance FromJSON A.TTransition
instance ToJSON A.Agent where
    toEncoding = genericToEncoding defaultOptions
instance FromJSON A.Agent
instance ToJSON A.Port where
    toEncoding = genericToEncoding defaultOptions
instance FromJSON A.Port
instance ToJSON A.ContextInfo where
    toEncoding = genericToEncoding defaultOptions
instance FromJSON A.ContextInfo
instance Store.Store A.Mode
instance Store.Store A.TTransition
instance Store.Store A.Agent
instance Store.Store A.Port
instance Store.Store A.ContextInfo
```

Listing 5.10: TestModel instance

5.2.7 ExploredService

This section discusses the Haskell class `ExploredService` and its implementations. The class is the most fundamental part of the LTS generation algorithm. The instance of the class is responsible for the accumulation of all the vertices and edges of the graph being explored. Additionally, the graph exploration algorithm checks if the given state is still *unprocessed* or was already *discovered*. So, the instance has to provide efficient look-up implementation. Storing constantly growing graph poses the key challenging problem for the graph exploration algorithm and becomes the bottleneck for the efficient exploration. After profiling the algorithm execution that will be described in the next chapter, the results show that `ExploredService` instance is responsible for most of the computation time and almost all the memory requirements of the algorithm operation. Those factors, however, highly depend on the concrete implementation. Therefore, the complexity will be discussed later on together with the specific implementation.

In Listing 5.11 the class code is presented. The first three lines provide definitions of the state identifier, which allows to uniquely identify the given state by the underlying container. A stored state is a tuple of state identifier and binary representation of the state, while a stored transition is a 3-tuple where source state is represented by its state identifier, transition is represented by its binary representation, and destination state is represented by its state identifier.

Next, a definition of the class `ExploredService` is introduced that defines three methods. All methods return results within an **`IO`** action to allow implementations having side effects. One of the possible side effects is a write operation to the file system.

The method `findAlreadyExploredStates` of the class provides the functionality to verify the state of the vertices: *undiscovered* vs *discovered*, cf. line 7. The method accepts, as its parameters, an instance of the service `(ExploredService a, State stateType, TTransition ↪ transitionType)=>a stateType transitionType` and a list of vertices accompanied by their hashes `[HashedState stateType]`. The implementation is expected to check all the state in the container and return them if found or return dummy value that represent missing value. Therefore, the method returns the list of stored states wrapped into **`Maybe`** instance within an **`IO`** action. When the vertex had already been discovered, the value constructor **`Just`** with the stored vertex is returned. Otherwise, the value constructor **`Nothing`** is returned.

The method `addExploredStates` provides the functionality to store a *discovered* vertices in a single batch, cf. line 9. Storing vertices in a batch allows leveraging batch store operation of the underlying container. The method accepts, as its parameters, the instance of the service `(ExploredService a, State stateType, TTransition transitionType)=>a ↪ stateType transitionType` and the list of vertices `[HashedState stateType]`. Once the state is inserted, it is guaranteed to be found by the previous method. The method returns the list of identifiers assigned to inserted states. The identifiers have to be stable after assignment.

The last method `addExploredTransitions` provides the functionality to store explored transitions, cf. line 11. The method accepts, as its parameters, the instance of the service `(ExploredService a, State stateType, TTransition transitionType)=>a ↪ stateType transitionType` and the list of edges represented as the identifier of the source vertex, identifier of the sink vertex and the label of the transition `[StoredTransition transitionType]`.

```haskell
1   type StateId = Int
2   type StoredState stateType = (StateId, stateType)
3   type StoredTransition transitionType = (StateId,transitionType,StateId)
4
5   class ExploredService a where
6
7       findAlreadyExploredStates :: (State stateType, TTransition transitionType) => a stateType transitionType −> [HashedState stateType]
            ↪ −> IO ([Maybe (StoredState stateType)])
8
9       addExploredStates :: (State stateType, TTransition transitionType) => a stateType transitionType −> [(StateHash,StoredState stateType)
            ↪ ] −> IO [StateId]
10
11      addExploredTransitions :: (State stateType, TTransition transitionType) => a stateType transitionType −> [StoredTransition
            ↪ transitionType] −> IO ()
```

Listing 5.11: ExploredService

All implementations store edges and vertices in separate data structures, rather than a single adjacency list. Storing edges in a separate data structure to vertices improves memory usage as there are only two contiguous storages rather than $\mathcal{O}(V)$ small lists. The adjacency lists allow to quickly identify all transitions that are reachable from the given state. However, from graph exploration perspective, it is not needed and therefore more compact data structure can be used.

The primary differentiating feature between proposed implementations is the amount of data stored in the main memory. There are hash-table based solutions that store all the states and transitions in the main memory. That approach provides the fastest access to data by both store and read operations, as main memory access is even 100 times faster than access to the hard drive [71]. The main disadvantage of this solution is that the memory consumption of the algorithm is high, and therefore the size of the model that can be successfully computed is the most constrained for that solution.

To limit the main memory usage of the algorithm, a solution that stores the vertices and edges in the database was proposed. There are two main advantages of the database-based solution. The first one is that it requires only constant: $\mathcal{O}(1)$ amount of main memory and linear: $\mathcal{O}(V)$ amount of disk memory, but the disk capacities are frequently two orders of magnitude higher than the main memory. So in theory, bigger problems can be explored. The second advantage is that all vertices and edges are easily quarriable during and after graph exploration, thanks to the SQL query language implemented by the SQL engine. So, this kind of solution is relatively the most generic. The main disadvantage is that the file-system access is slow, and the layer of creating the queries to the database make it even worse. Databases are optimised for the reliable data persisting. Therefore, any committed transaction has to be stored reliably, i.e. data has to be flushed into the disk, which negates buffers advantages. The databases keep data quarriable and therefore additional data structures (like database indexes) have to be updated when a write operation to the table is performed. So, the slowness of the hard disk access is magnified by the additional operations made by the database.

The last proposed solution tries to leverage advantages of both previous solutions. The vertices and edges are stored in two separate dedicated append-only files. Both vertices and edges are stored in their files in the order of their discovery, with no additional structure. Therefore, store operation is more optimised as appending into the file has constant complexity: $\mathcal{O}(1)$, and does not require additional overhead of updating indexes or parsing an SQL query. Moreover, the data could be buffered and stored only in bigger chunks. This optimises access to the hard drive. As the data are stored in the permanent storage, they can be removed from main memory. However, as the graph generation algorithm requires testing if the vertex is already discovered, that would require a linear scan of the file. The read operation in the file without the structure soon will become a bottleneck of the LTS generation algorithm. To speed up the operation `findAlreadyExploredStates` without the impact on the store operation, the set of already discovered states is kept in the main memory. To save even more main memory, instead of complete state, only vertices hashes are kept in main memory. Therefore, the main memory requirement of this hybrid solution is again $\mathcal{O}(V)$ (like in the case of hash tables), but in this case, the constant is smaller is significantly smaller. Therefore, systems with intermediate number of states can be calculated but faster than in the case of database storage.

Let us see the implementation that supports the first solution – all data stored in main memory. In Haskell, there are various implementations of the dictionary (associative table). The most promising implementation was chosen based on results of an online benchmark [56]. The implementation comes with two flavours, as in Haskell multiple variants of the same data structure may be available.

The instances `InMemoryGraph` and `InMemoryGraphIO` store both vertices and edges into two separate hash tables [48, 158, 95]. An instance `InMemoryGraph` implementation of the class `ExploredService` is presented in Listing 5.12.

Based on the presented examples, it is relatively easy to add another implementation that would depend on a different library.

```
type VisitedStates s stateType = HashTable s StateHash (StoredState stateType)
type VisitedTransitions s transitionType = HashTable s TransitionHash transitionType
type TransitionHash = (StateId, StateId)

type HashTable s k v = H.HashTable s k v

data InMemoryGraph stateType transitionType = InMemoryGraph (VisitedTransitions RealWorld transitionType) (VisitedStates RealWorld
    ↪ stateType)

mkInMemoryGraph = do
  visitedStates <− stToIO H.new
  visitedTransitions <− stToIO H.new
  return (InMemoryGraph visitedTransitions visitedStates)

instance ExploredService InMemoryGraph where

    −−findAlreadyExploredStates :: a −> [HashedState] −> IO ([Maybe StoredState])
    findAlreadyExploredStates (InMemoryGraph transitions states) statesToFind = do
        let statesToFindHashes = map fst statesToFind
        mapM (stToIO . H.lookup states) statesToFindHashes

    −−addExploredStates :: a −> [(StateHash,StoredState)] −> IO [StateId]
    addExploredStates (InMemoryGraph transitions states) statesToAdd = do
        let ids = map (fst . snd) statesToAdd
        mapM_ (insertInto states) statesToAdd
        return ids
        where
            insertInto st (k,v)= stToIO (H.insert st k v)

    −−addExploredTransitions :: a −> [StoredTransition] −> IO ()
```

```
addExploredTransitions (InMemoryGraph transitions states) transitionsToAdd = do
    let toInsert = map (\ (from,tr,to)−>((from,to),tr) ) transitionsToAdd
    mapM_ (insertInto transitions) toInsert
    where
        insertInto tr (k,v)= stToIO (H.insert tr k v)
```

Listing 5.12: InMemoryGraph

Let us discuss another implementation that supports the database solution – all data stored on hard drive disk by a database management system. There are many relational database management systems (RDBMS) available. Here, the implementation using simple and lightweight version of RDBMS was chosen, namely SQLite. The main advantage of SQLite over different RDBMS is that it is a self-contained, serverless library. Therefore, it does not require any additional software being installed apart from the standard ODBC library. The queries to the database are executed within the main thread of the application itself, rather than making an additional network call to the database server.

In Listing 5.13, an instance of the class `ExploredService` is presented that implements database operations. In the first line, the definition of the tables is provided using template Haskell extension for each table the respective Haskell data type will be generated as well. Next, the definition of the data type that represents the container is defined. The container contains the instance of database connection pool that allows to communicate with the database and three methods that allow manipulation of the data. The first function `findByHash` is a factory method that enables creation of the SQL query that looks up states by their hash. Two later methods, `toPersistSystemState` and `toPersistTransition` allow transformation of vertices and edges into database-compliant format (data transfer objected created in the previously). `mkPooledSqlStorage` is the factory methods that allows creation of instances of the `SqlGraphPooledStorage` container. The function accepts three parameters: an algorithm name used to serialize states and transitions (can be one of JSON or STORE), the database file name, and additional settings used to create a database. The function is responsible for creation of a connection pool, running migration – SQL scripts to create required tables, creation of additional index to allow fast look up of vertices by their hash, and proper initialisation of the data container instance. The latter declared functions help to perform the above subtasks.

`SqlGraphPooledStorage` instance implementation is straightforward: `addExploredStates` ↪ uses `insertEntityMany`, and `addExploredTransitions` uses `insertMany` which perform batch addition of the entities into the database. Vertices and transitions are transformed to data transfer objects generated by template Haskell code using helper methods provided by the container. `findAlreadyExploredStates` takes advantage of the function `findByHash` defined in the container. `findByHash` method performs a select statement by state hash and returns the first state found.

As there was defined index on state hash column, finding a state by its hash has logarithmic complexity.

Inserting new state and transition has logarithmic complexity as well as it requires updating the indexes.

```
share [mkPersist sqlSettings, mkMigrate "migrateAll"] [persistLowerCase|
ToPersistSystemState
    index Int
    hash StateHash
    systemState ByteString
    UniqueHash hash
    deriving Show

ToPersistTransition
    fromSystemState Int
    toSystemState Int
    transition ByteString
    UniqueTransition fromSystemState toSystemState
    deriving Show
|]

data SqlGraphPooledStorage stateType transitionType =
  SqlGraphPooledStorage {
    pool::ConnectionPool,
    findByHash::(StateHash -> SqlQuery DefaultLogging (Maybe (StoredState stateType))),
    toPersistSystemState :: (StateHash,(StateId,stateType)) -> ToPersistSystemState,
    toPersistTransition:: (StateId, transitionType, StateId) -> ToPersistTransition}

mkPooledSqlStorage :: (State stateType, TTransition transitionType) =>
  String -> Text -> [Text] -> IO (SqlGraphPooledStorage stateType transitionType)
mkPooledSqlStorage serializationAlgorithm dbName' pragmas' = do
  pool <- mkPoolWithPragmas dbName' pragmas'
  runPooledDbSchema pool $ do
    runMigration migrateAll
    rawExecute "CREATE␣INDEX␣hash_index␣ON␣to_persist_system_state(hash)" []
  return $ pooledStorage pool serializationAlgorithm

mkPoolWithPragmas :: Text -> [Text] -> IO (Pool SqlBackend)
mkPoolWithPragmas dbName' pragmas = mkPool info 1
  where
    info = fromStrict $ E.encodeUtf8 $
      "{\"connectionString\":␣\"" `append` dbName' `append`
      "\",␣\"walEnabled\":␣false,␣\"fkEnabled\":␣false,␣\"extraPragmas\":␣"
      `append` tshow pragmas `append`"}"
    tshow = pack . show

mkPool :: LBS.ByteString -> Int -> IO (Pool SqlBackend)
mkPool info nConnections = runDefaultLoggingT $ createSqlitePoolFromInfo info' nConnections
  where
    Just info' = decode info

pooledStorage pool "JSON" = SqlGraphPooledStorage pool (findByHash fromJsonToPersistSytemState) toJsonToPersistSystemState
        ↪ toJsonToPersistTransition
```

```
pooledStorage pool "STORE" = SqlGraphPooledStorage pool (findByHash fromStoreToPersistSytemState) toStoreToPersistSystemState
      ↪ toStoreToPersistTransition

findByHash :: State stateType => (ByteString -> stateType) -> StateHash -> SqlQuery DefaultLogging (Maybe (StoredState stateType))
findByHash fromEncodedToPersistSytemState stateHash = do
  let dbHash = B.fromShort stateHash
  entityM <- selectFirst [ToPersistSystemStateHash ==. dbHash] []
  return $ fmap (toStoredState fromEncodedToPersistSytemState) entityM



instance ExploredService SqlGraphPooledStorage where
  --findAlreadyExploredStates :: a -> [HashedState] -> IO (ReaderT [Maybe StoredState])
  findAlreadyExploredStates (SqlGraphPooledStorage pool findByHash _ _) statesToFind = runPooledDbQuery pool $ do
    let statesToFindHashes = map fst statesToFind
    mapM findByHash statesToFindHashes

  --addExploredStates :: a -> [(StateHash,StoredState)] -> IO [StateId]
  addExploredStates (SqlGraphPooledStorage pool _ toToPersistSystemState _) statesToAdd = runPooledDbUpdate pool $ do
    let ids = map (fst . snd) statesToAdd
    let toInsert = map toToPersistSystemStateWithKey statesToAdd
    insertEntityMany toInsert
    return ids
    where
      toToPersistSystemStateWithKey entity =
        Entity (getKey entity) (toToPersistSystemState entity)

  --addExploredTransitions :: a -> [(StateId,StoredTransition,StateId)] -> IO ()
  addExploredTransitions (SqlGraphPooledStorage pool _ _ toToPersistTransition) transitionsToAdd = runPooledDbUpdate pool $ do
    let toInsert = map toToPersistTransition transitionsToAdd
    insertMany toInsert
    return ()

getKey :: State stateType => (StateHash, StoredState stateType) -> Key ToPersistSystemState
getKey (_,(stateId,_)) = toSqlKey (fromIntegral stateId)
```

Listing 5.13: SqlGraphPooledStorage

In Listing 5.14, an instance of the class `ExploredService` is presented that implements an in-memory cache for vertices. The instance is used together with `SqlGraphPooledStorage` as a decorator pattern. That means that cache implementation intercepts invocation of particular methods, performs additional processing and decides to invoke underlying container implementation. The calls are only in a way that e.g. `addExploredTransitions` of `HashedTrieCached` may call `addExploredTransitions` of underlying storage and no other methods of that container.

The cache is intended to store only states in memory up to certain number. Therefore, `addExploredTransitions` just forwards call to underlying container and does not do any additional computations. The methods that are responsible for retrieving and storing states however try to leverage the cache. So `addExploredStates` stores states in the cache and in the underlying container. Storing states in the cache makes them available for look up later on and its storing implementation performs automatic eviction of the least recently used items. Method `findAlreadyExploredStates` tries to find the states in the cache and if the data there are missing it reaches to the underlying container. Trie lookup and insert complexity is linear in the size of the key used to encode the data and is not correlated with the size of the data.

```haskell
type DefaultCache stateType = DefaultCacheST RealWorld stateType
type DefaultCacheST s stateType = STRef s (Cache.HashedTrie StateHash (StoredState stateType))



data HashedTrieCached sqlStorage stateType transitionType =
  HashedTrieCached { cache :: DefaultCache stateType, storage:: sqlStorage stateType transitionType}

mkCached :: (State stateType, TTransition transitionType, ExploredService sqlStorage) =>
  Int -> sqlStorage stateType transitionType -> IO (HashedTrieCached sqlStorage stateType transitionType)
mkCached cacheSize exploredService = do
  cache <- newCache cacheSize
  return (HashedTrieCached cache exploredService)

instance ExploredService storage => ExploredService (HashedTrieCached storage) where
  --findAlreadyExploredStates :: a -> [HashedState] -> IO (ReaderT [Maybe StoredState])
  findAlreadyExploredStates (HashedTrieCached cache storage) statesToFind = do
    cachedValues <- mapM (\ (k,_) -> lookupIO k cache) statesToFind
    let zipped = zip statesToFind cachedValues
    let hit = filter (isJust . snd) zipped
    let missed = filter (not . isJust . snd) zipped
    doubleLookedUp <- findAlreadyExploredStates storage (map fst missed)
    return (map snd hit ++ doubleLookedUp)

  --addExploredStates :: a -> [(StateHash,StoredState)] -> IO [StateId]
  addExploredStates (HashedTrieCached cache storage) statesToAdd = do
    mapM_ (\ (k, v) -> insertIO k v cache) statesToAdd
    addExploredStates storage statesToAdd

  --addExploredTransitions :: a -> [(StateId,StoredTransition,StateId)] -> IO ()
  addExploredTransitions (HashedTrieCached cache storage) transitionsToAdd = do
    addExploredTransitions storage transitionsToAdd


newCache :: State stateType => Int -> IO (DefaultCache stateType)
newCache cacheSize =
    stToIO (newSTRef (Cache.empty cacheSize))

insertIO :: State stateType => StateHash -> StoredState stateType -> DefaultCache stateType -> IO (Maybe (StateHash, StoredState
     ↪ stateType))
insertIO k v cache =
    (stToIO . insertST k v) cache

lookupIO :: State stateType => StateHash -> DefaultCache stateType -> IO (Maybe (StoredState stateType))
lookupIO k cache =
    (stToIO . lookupST k) cache

insertST :: State stateType => StateHash -> StoredState stateType -> DefaultCacheST s stateType -> ST s (Maybe (StateHash, StoredState
    ↪ stateType))
insertST key value cacheRef = do
    c <- readSTRef cacheRef
    let (modifiedCache, removed) = Cache.insert key value c
    writeSTRef cacheRef modifiedCache
    return removed

lookupST :: State stateType => StateHash -> DefaultCacheST s stateType -> ST s (Maybe (StoredState stateType))
lookupST key cacheRef = do
    c <- readSTRef cacheRef
    let (modifiedC, found) = Cache.lookup key c
    writeSTRef cacheRef modifiedC
    return found
```

Listing 5.14: HashedTrieCached

The last example presents the implementation of the hybrid solution. Hybrid solution stores states and transition in a simple unstructred file on the disk. In order to provide fast lookup it requires augmentation with an in-memory data structure that keeps track of discovered states.

In Listing 5.15 an instance `FileStore` is presented. The implementation stores both states and transitions in two separate files on the hard drive. The data are appended to the file only which makes the complexity of the insertion constant. `addExploredStates` and `addExploredTransitions` methods are performing buffered append operation on the file leveraging `hPut` library method. The buffered append operation stores data from consequential invocations into a in-memory buffer and performs a single disk write operation only when the buffer is full. As the file on the disk has no additional structure, searching through the file would have linear complexity and therefore it would be highly inefficient. Therefore, `findAlreadyExploredStates` method was left unimplemented. In order to enable fast look up of the states the container is intended to be used only with an additional in-memory storage. Storing items in the file is buffered to decrease number of slow disk writing operations and take advantage of relatively fast batch writes. `mkFileStore` is a factory method that initialises container. It expects two file paths as its argument and two methods that will allow seralization of generic transition and generic state respectively. The first file will used to store states in the disk and the second file is intended to contain explored trasitions. the factory method is responsible for opening both files in write mode and setting buffers for writing operation to 128kB. Lastly, container is returned initialised with enconding functions and handlers to opened files.

Keeping encoder method instances in the container, rather than using hard coded encoder, allows injection of various serialisation methods during initialisation of the container. In that approach different serialisation methods can be tested and it does not require changing container code.

```
type Encoder a = a -> BS.ByteString
data FileStore stateType transitionType =
  FileStore (Encoder (StoredTransition transitionType)) (Encoder (StoredState stateType)) IO.Handle IO.Handle

mkFileStore :: (State stateType, TTransition transitionType) =>
  FilePath -> FilePath -> Encoder (StoredState stateType)-> Encoder (StoredTransition transitionType) -> IO (FileStore stateType
      ↪ transitionType)
mkFileStore stateFile transitionFile stEncoder trEncoder= do
    visitedStates <- IO.openFile stateFile IO.WriteMode
    IO.hSetBuffering visitedStates (IO.BlockBuffering (Just (128*1024)))
    visitedTransitions <- IO.openFile transitionFile IO.WriteMode
    IO.hSetBuffering visitedTransitions (IO.BlockBuffering (Just (128*1024)))
    return (FileStore trEncoder stEncoder visitedTransitions visitedStates)

instance ExploredService FileStore where
    --findAlreadyExploredStates :: a -> [HashedState] -> IO ([Maybe StoredState])
    findAlreadyExploredStates (FileStore _ encode transitions states) statesToFind = do
        error "Not␣implemented"

    --addExploredTransitions :: a -> [StoredTransition] -> IO ()
    addExploredTransitions (FileStore encode _ transitions states) transitionsToAdd = do
        mapM_ (putInto transitions) transitionsToAdd
        where
            putInto transitionFileHandler transition = BS.hPut transitionFileHandler (encode transition)

    --addExploredStates :: a -> [(StateHash,StoredState)] -> IO [StateId]
    addExploredStates (FileStore _ encode transitions states) statesToAdd = do
        let ids = map (fst . snd) statesToAdd
        mapM_ (putInto states) statesToAdd
        return ids
        where
            putInto stateFileHandler (_,state) = BS.hPut stateFileHandler (encode state)
```

Listing 5.15: FileStore

In Listing 5.16 a complementary instance to the `FileStore` is presented which is called `InMemoryHashesOnly`. Like the name of the implementation suggests, the container is responsible for storing hashes of the states in main memory. Main memory guarantees low latency look up time and storing only hashes allows to lower the memory footprint of the algorithm. The instance follows a decorator pattern and provides efficient implementation of `findAlreadyExploredStates` operation. `addExploredStates` and `addExploredTransitions` do not add any value and merely forward insertion operations into the underlying container.

As look up of the states in the unstructured file is highly inefficient due to both algorithmic complexity and latecy with access to the file on hard drive, the in memory data structure is used to store the states. Additionally, look up operation requires only the information of the presence of the state in the container, therefore, only hashes of the states are stored to save additional memory. The states are not lost as they are stored in the disk file. Using only hashes to search for the explored states assumes that hashes of the state are unique. If that assumption is not correct, some states of the state space can be left unexplored due to hash collision and false positive feedback of already explored state information. Uniqueness of hashes cannot be guaranteed in general but for feasible sizes of state space the probability of hash collisions can be negligible for certain hash functions.

```haskell
data InMemoryHashesOnly anyStorage stateType transitionType =
  InMemoryHashesOnly (anyStorage stateType transitionType) (H.BasicHashTable StateHash StateId)

mkInMemoryHashesOnly storage = do
  visitedStatesHashes <− H.new
  return $ InMemoryHashesOnly storage visitedStatesHashes

instance ExploredService storage => ExploredService (InMemoryHashesOnly storage) where
  −−findAlreadyExploredStates :: a −> [HashedState] −> IO (ReaderT [Maybe StoredState])
  findAlreadyExploredStates (InMemoryHashesOnly storage hashes) statesToFind = do
    cachedValues <− mapM (\(k,_) −> H.lookup hashes k) statesToFind
    let zipped = zip statesToFind cachedValues
    let found = map (\((h,s),mId) −> fmap (\i−> (i,s)) mId) zipped
    return found

  −−addExploredStates :: a −> [(StateHash,StoredState)] −> IO [StateId]
  addExploredStates (InMemoryHashesOnly storage hashes) statesToAdd = do
    mapM_ (\(k, (v,_)) −> H.insert hashes k v) statesToAdd
    addExploredStates storage statesToAdd

  −−addExploredTransitions :: a −> [StoredTransition] −> IO ()
  addExploredTransitions (InMemoryHashesOnly storage _) transitionsToAdd = do
    addExploredTransitions storage transitionsToAdd
```

Listing 5.16: InMemoryHashesOnly

5.2.8 defaultLtsIO

The last missing piece of the LTS graph generation algorithm is the initialisation of the main loop that controls graph exploration and steers the computation. The main loop as explained in Sec. 5.2 is a realisation of DFS or BFS algorithms. The variant of the algorithm is determined by the internal data structure used to keep track of unexplored vertices of the LTS graph that represent state space points. An example initial setup of data structures `defaultLtsIO` and initialisation of the main loop `ltsIO` is shown in

Listing 5.17. At the listing an example composition of data structures is shown, here queue is an in-memory queue (cf. Sec. 5.2.4), hashing algorithm is an SHA256 cryptographic hash (cf. Sec. 5.2.5), and storage of the graph is flushed to the files on the disk and keeping only hashes of the states in main memory (cf. Sec. 5.2.7). Moreover, `defaultLtsIO` accepts an initial state and a generic Alvis model definition. The second presented method `ltsIO` adds initial state to the explored service, next it puts the initial state to the queue, and finally calls the main loop with all initialised data structures. Because the defaultLtsIO method is set up in the example with a queue, the presented algorithm represents a breadth-first search variation of graph exploration.

```
defaultLtsIO :: (
  State stateType,
  TTransition transitionType,
  GraphDescription model) =>
    stateType ->
    model stateType transitionType -> IO (srv stateType transitionType)
defaultLtsIO initialState model = do
  queue <- mkInMemoryQueue
  hasher <- mkSHA256Hasher
  storage <- mkFileStore "states.dat" "transitions.dat" Store.encode Store.encode
  service <- mkInMemoryHashesOnly storage
  ltsIO initialState queue hasher service model
  return service

ltsIO :: (
  ToVisitService toVisitSrv,
  Hasher h,
  ExploredService exploredSrv,
  State stateType,
  TTransition transitionType,
  GraphDescription model) =>
    stateType ->
    toVisitSrv stateType transitionType ->
    h ->
    exploredSrv stateType transitionType ->
    model stateType transitionType -> IO ()
ltsIO initialState toVisitService hasher exploredService model = do
    idGenerator <- mkIdGenerator
    addExploredStates exploredService [((stateHash hasher initialState),(0,initialState))]
    scheduleStatesToVisit toVisitService [(0,initialState)]

    loop $ ltsComputationLoopIO toVisitService hasher exploredService idGenerator model
    return ()
```

Listing 5.17: defaultLtsIO

5.3 Partial order reduction

Partial order reduction is a technique that tries to reduce the number of states of state space. By discarding states that are not meaningful from a model checking perspective, one can significantly lower the number of states stored by the algorithm. The topic here is just mentioned here for completeness perspective, but it is not a goal of this dissertation to fully explore all possibilities. Therefore, the POR reduction would be introduced informally in this paragraph. The main point is to show that partial order reduction can be added to the graph generation algorithm without modification of the main exploration algorithm.

Alvis has that property that agent's local variables are modified only by instructions of that particular agent and any exchange of the variables between two different agents may happen only through communication instructions. Moreover, the state of the agent can be modified by another agent only by start, and exit instructions. Additionally, there is as well an instruction `pick` that changes the state of the current agent only, but it introduces non-determinism in the system – there is more than one transitions enabled for that particular agent. All other instruction, let us call them simple instructions, affect the state of an agent that executed that instruction and have only one subsequent state. Therefore, if model checking task is to check if system is deadlock-free, one can compress a sequence of states related to execution of simple instructions, and instead of computing the very next state, the whole path within the state graph can be computed in memory, and only the last state of the path may be stored.

Introduction of the path compression can be achieved by providing a wrapper for `fire` function which is shown in Listing 5.18 as `fireWithPor`. Instead of just firing the given transition for the given state, algorithm explores a path for that particular agent starting from the given state and pointed by transition. The path computation is ended if the enabled transition is not a simple transition verified by predicate `isNotSimpleTransition`. Path compression calculation is happening in the recursive helper function `toFinalTransitions`. The method stops when there is no transition enabled, it stops as well when there are more than one transition enabled (e.g. non-deterministic pick instruction), or the transition is matched by the predicate, and it is not the simple one. There is an additional condition to stop computations if the agent executes some code in the loop with only simple instructions. Then, if the computations reach the initial state, it stops. Therefore, if agent may execute **`exec`** instruction then `if`, then another **`exec`** and finally **`out`** instruction. The result of execution `fireWithPor (TExec Agent 1)(X,1,[],(1,2)` ↪ `)Agent` instead of advancing agent to the next state will compute all four transitions and the resulting state will be returned e.g. `(W,4,[out p],(6,0))`.

```
fireWithPor :: TTransition -> State -> Agent -> [State]
fireWithPor transition state agent = concat (map (\tr -> fire tr newState) newTransitions)
  where
    (newState, newTransitions) = toFinalTransitions agent state [transition] state

```

```
toFinalTransitions :: Agent -> State -> [TTransition] -> State -> (State,[TTransition])
toFinalTransitions agent state [] _ = (state,[])
toFinalTransitions agent state [tr] startState
        | isNotSimpleTransition tr = (state,[tr])
        | newState == startState = (newState,[])
        | otherwise = toFinalTransitions agent newState (enable newState agent) startState
          where
             [newState] = fire tr state
toFinalTransitions agent state trs _ = (state, trs)


isNotSimpleTransition :: TTransition -> Bool
isNotSimpleTransition (TDelay _ _) = False
isNotSimpleTransition (TExec _ _) = False
isNotSimpleTransition (TExit _ _) = True
isNotSimpleTransition (TIf _ _) = False
isNotSimpleTransition (TIn _ _) = False
isNotSimpleTransition (TInAP _ _ _) = True
isNotSimpleTransition (TInPP _ _ _) = True
isNotSimpleTransition (TInF _ _ _) = True
isNotSimpleTransition (TJump _ _) = False
isNotSimpleTransition (TLoop _ _) = False
isNotSimpleTransition (TLoopEvery _ _) = False
isNotSimpleTransition (TNull _ _) = False
isNotSimpleTransition (TOut _ _) = True
isNotSimpleTransition (TOutAP _ _ _) = True
isNotSimpleTransition (TOutPP _ _ _) = True
isNotSimpleTransition (TOutF _ _ _) = True
isNotSimpleTransition (TSelect _ _) = False
isNotSimpleTransition (TStart _ _) = True
isNotSimpleTransition (STInAP _ _ _) = True
isNotSimpleTransition (STInPP _ _ _) = True
isNotSimpleTransition (STOutAP _ _ _) = True
isNotSimpleTransition (STOutPP _ _ _) = True
isNotSimpleTransition (STDelayEnd _ _) = False
isNotSimpleTransition (STLoopEnd _ _) = False
isNotSimpleTransition (STInEnd _ _) = False
isNotSimpleTransition (STOutEnd _ _) = False
isNotSimpleTransition (STTime _) = False
isNotSimpleTransition (TNone) = False
```

Listing 5.18: Partial order reduction in Alvis model

Chapter 6

Results

This chapter discusses the evaluation of the algorithm proposed in the previous chapter 5. Various tests are run, and the implementation was adjusted when some problems were signalized during the testing phase. The tests that were performed as an evaluation of the work are the following:

- profiling the application with the native GHC capabilities, i.e., the fraction of time required for specific parts of the code.
- profiling the memory usage by the application and the specific functions and data structures with the native GHC capabilities.
- monitoring the system resources usage by the GNU program *time*.
- monitoring execution time by the Haskell library *criterion*.
- creation of the Alvis models family that easily scale-up the number of states in the model. By scaling-up, it is understood that one can create another model with more states in its state space by changing one global parameter. Additionally, it is easy to calculate their number of states without running the model.
- comparing applied optimisations by measuring usage of resources by reference implementation and optimised one.
- comparing optimised algorithm with state of the art model checker.

6.1 Problems encountered during the implementation

During the implementation and testing phases of the algorithm development, several problems were encountered. Most significant ones were mentioned in the list below:

- Compilation time exceeding one hour is frequent. Haskell is a native compiled language, and therefore before running the program, one has to download all the libraries and compile them together with the source code of the program. Therefore, the initial compilation of a medium-sized project is exceptionally long compared to non-native languages like Java or interpreted languages like Python.

- Cabal *Common Architecture for Building Applications and Libraries* [93] is one of a default system for building and packaging Haskell libraries and programs. However, the project build with cabal tool is unstable because of the implicit reliance on the version of the GHC installed on the development machine. Contrary to other programming languages, the version of the GHC implies the version of the standard library, and therefore most third-party libraries have to have a specific version. Once the GHC installation is updated, the build can fail because of unresolvable dependencies. Cabal build script does not allow to specify the version of the GHC required.
- Haskell package repository is open for anyone to contribute his/her library. As a result, some libraries were created a while ago and are no longer maintained. Therefore, they work only with some specific old version of the GHC (e.g. *loop-while, bitset*), they do not implement the newest features, (e.g. xxHash does not support 128-bit nor 64-bit hashing while its reference C implementation does), or libraries having conflicting dependency lists.
- Stack is a build tool for Haskell that tries to fix problems of Cabal, GHC and other library version conflicts. The stack is built on top of the Cabal. The build with stack uses Hackage and Stackage lists of tested libraries that are guaranteed to be compatible with each other. However, changing the resolver version requires downloading all the dependencies together with the specific version of the GHC, which takes even more time.
- By default, stack build requires to recompile all the modules after the build is changed between targets: development, benchmarking, profiling or production.
- Various implementation of standard data structures coexist in multiple libraries in the Haskell package system.
- The libraries' documentation is frequently unaccompanied with examples that make it harder to use. To understand the behaviour of the code, one has to investigate the source code of the library.
- Even though Haskell is equipped with the garbage collector, due to the open bug in the `ByteString` library, the graph generation algorithm was affected by memory leakage [73].
- The library providing an implementation of a specific data structure comes in various variants, as in Haskell code can be: lazy vs strict, immutable vs mutable, boxed vs unboxed. The default Haskell choice is a code and data structures that are lazy, immutable and that use boxed types.
- Lazy code evaluation is a core feature of Haskell runtime, but it comes with a trade-off of additional memory required to keep track of the evaluations. Additionally, it makes reasoning about the time complexity and control over the memory usage harder. Finally, when trying to optimise the code, it usually produces less optimal code than strict code evaluation.
- The purely functional (immutable) data structures have from time to time some performance issues

over its imperative (mutable) counterparts. Again, during optimisation of the code, usually it is better to use non-default mutable data structures.

- Haskell is missing any standard way to dynamically load other libraries (after the application was compiled, when the program is running without the recompilation), therefore testing various data structures and models requires the addition of the library or model to the project directly and its recompilation.

Most of the problems mentioned above were addressed and solved, but the solutions have some disadvantages as well. The list of steps taken are as listed:

- Moving the project to use stack toolchain resolved the problem with the GHC version because stack downloads specified version of the compiler for the project.
- The stack tool requires to download many dependencies, but it is done sporadically when the project is set up on the development machine, or the version of the compiler is changed in the project requirements.
- There is an option to put the separate builds of development, testing, benchmarking, and production into different directories. This option is not handled by default and user need. Once used, the recompilation process is not performed when different builds are one after another.
- In case of *loop-while* library, the third-party library's dependencies requirements were too strict, and the library was used in the project after downloading the library manually, adjusting its build script and dependencies, and adding it into the project as a local dependency. This was achievable as the library was relatively small, just a few tens lines of code and a few dependencies.
- The choice of the proper data structure library implementation from the competing list of libraries was based on the benchmarks found online [56].
- The bug in the `ByteString` was addressed by an additional memory copy from `ByteString` into `ShortByteString`. An additional copy of the memory allowed freeing 1.5 GB of memory in the case of M1048K model $y = 16$.
- In Haskell, there are impure data structures and bindings to the C libraries. It requires more testing and reading [28].
- Lazy evaluation can be switched off in some libraries, making the code easier to understand and less main memory consuming.
- Regarding the loading code dynamically at runtime into a Haskell application, there exists a third-party library created by Facebook [44], but it requires specially adjusted compiler, and therefore to avoid further complications, the approach to add more code into the application was chosen.

6.2 Test model

To test the performance of the graph generation algorithm, the simple parametric model was designed. The model was designed to be simple and easily scalable to reach different state space sizes. The model design allows accurate calculation of all reachable states. The model includes four active agents named $A1,\ldots,A4$. Every agent in the model behaves in the same way. The agents do not communicate with each other. Therefore, the state space of the whole system is a Cartesian product of the state space of a single agent. Lack of communication between agents removes any synchronisation between agents, and therefore, the state space is symmetric. The graphical layer of the test model is presented in Fig. 6.1. Every active

Figure 6.1: Graphical layer of test model

agent has two variables x, and y. The variable x is a counter which is incremented by one. The variable y is constant for the given model instance, but it is a meta-parameter for the family of the models. The variable y constrains the maximal possible value of the variable x when the latter is incremented. The model behaviour is infinite, but the state space is finite. Every agent is able to perform two instruction: unconditional, infinite **`loop`**, and **`exec`** that increments x by one modulo y. The second instruction is executed within the loop. The code layer for the test model and the value of the parameter $y = 10$ is presented in Listing 6.1. The number of states in the model n_s depends on the value of the parameter of the model y and can be calculated as:

$$n_s = (2y)^4 \tag{6.1}$$

The number of states of a single agent is $2y$. The agent has two instructions, and therefore its program counter has two possible values of 1 and 2. The agent mode is always X – running. The context list of the agent is always empty. For the given value of the parameter of the model y, only variable x changes its value from 0 up to $y-1$, and therefore it has y different values. The pc takes both of its value for every value of x. Therefore, the number of states is $2y$. Finally, as agents do not depend on each other, the number of states for the system of k agents is $(2y)^k$. The test model has arbitrarily fixed the number of agents to $k = 4$.

```
agent A1, A2, A3, A4 {
  x :: Int = 0;
  y :: Int = 10;

  loop {
    exec x = (x + 1) ‘mod‘ y;
  }
}
```

Listing 6.1: Simple scalable test model

The number of transitions in the model n_t depends on the value of the parameter of the model y and can be calculated as:

$$n_t = 4 \cdot n_s = 4\,(2y)^4 \tag{6.2}$$

This is because, in every state of the system, there are always enabled four instructions to be executed, one for every agent.

The subset of the LTS graph for the test model and the parameter set to $y = 1$ is presented in Fig. 6.2. The whole system consists of 16 states and 64 transitions. Out of 16 original states, only six states are shown explicitly as in the Alvis compiler's output. Out of 64 original transitions, only 24 transitions are shown explicitly as in the output of the Alvis compiler. Other directly reachable states are presented only as numbers to increase the readability of the diagram. The transitions that start from the states depicted with the only number were removed as well the states 13, 14, and 15. In the diagram, one can verify that transition from the system state 0 to the system state 1 is related to agent $A1$ executing its `loop` instruction. The program counter of the first agent is therefore changed to 2. The transition from the system state 1 back to the system state 0 is related to the agent $A1$ executing its `exec` instruction. Incrementing modulo 1 does not change its variable state, and its program counter returns to 1. The transition from the system state 1 to the system state 5 is related to the agent $A2$ executing its `loop` instruction. Therefore, two agents have their program counter set to 2.

Increasing the value of the model parameter y increases the number of states that system has according to the formula:

$$\text{number of states} = (2y)^4 \tag{6.3}$$

In Table 6.1, the value of parameter y and the size of the state space is collected.

6.3 Testing environment

The algorithm performance was measured on a PC class computer. In this section, the basic parameters of the testing machine are described. The testing machine has a Debian GNU/Linux 10 (buster) operating

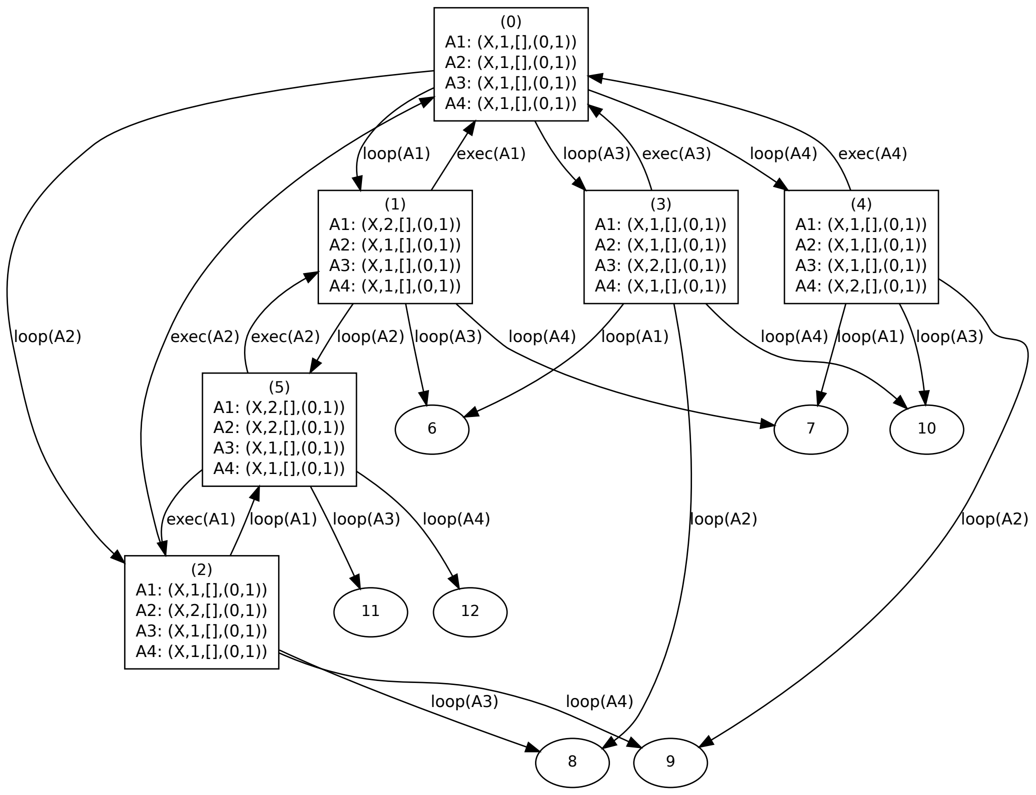


Figure 6.2: Subset of LTS graph of the test model for y = 1. Only 6 out of 16 states.

Table 6.1: Number of states and transitions depending on the value of parameter y for the test model.

y	number of states	y	number of states	y	number of states
1	16	7	38,416	29	11,316,496
2	256	8	65,536	50	100,000,000
3	1,296	9	104,976	63	252,047,376
4	4,096	10	160,000	89	1,003,875,856
5	10,000	16	1,048,576	159	10,226,063,376
6	20,736	28	9,834,496	282	101,185,065,216

system (Linux distribution). The result of the uname program executed on the testing machine shows the version of the operating system and Linux kernel version:

```
$> uname -v
#1 SMP Debian 4.19.146-1 (2020-09-17)
```

The computer is equipped with the following hardware:

- CPU – Intel Core i5-9400 CPU @ 2.90GHz (6 cores, no Hyper-Threading, 2.90 GHz base frequency, and 4.10 GHz max frequency)
- main memory – 32 GB DDR4 RAM
- presistant memory – SSD, 256 GB, M.2, sequential write speed up to 2200 MB/s

In order to measure time performance of the algorithm, the Haskell Criterion library is used [140]. The library allows preparing the benchmarks that run the tested application in several iterations. Every iteration consists of one, two, three, and more runs of the application. Total time of iteration, together with the time of a single run, is measured. The linear regression model is then fitted to the dataset composed of the iteration vs total time of computation for an iteration. The example of the fitting can be seen in the graph generated by the Criterion package in Fig. 6.3. Based on the fitted model, the execution time can be estimated as a slope parameter of the regression model. Secondly, the average of all the runs is calculated, and the errors are estimated based on the statistical bootstrap method.

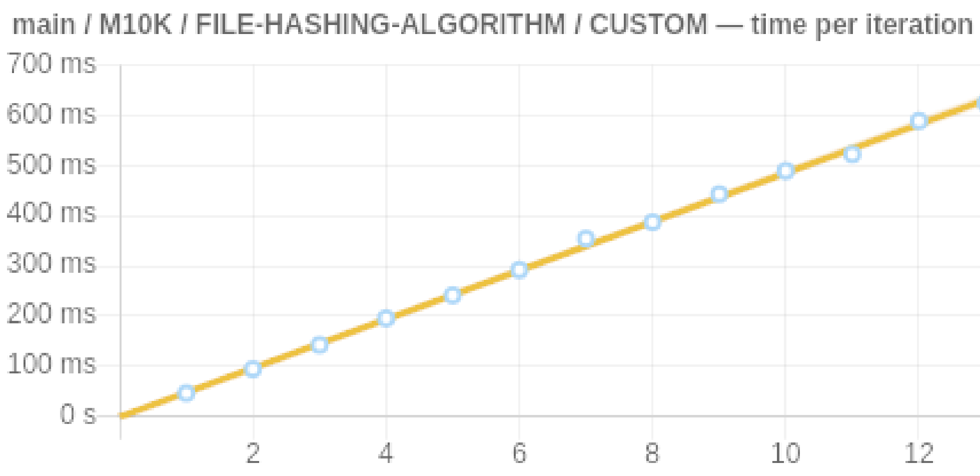


Figure 6.3: Citerion results of linear regression fitting to total computation time of iterations. i^{th} iteration consists of i runs of the algorithm. Model $y = 5$ of 10000 states. Algorithm generation stores the states and transtions into files on the disk, in main memory only hashes of the states, hashes are computed with CUSTOM hasher.

The Criterion package does not allow to measure the maximum resident memory size. In order to measure memory consumption by the algorithm, the GNU *time* program was used. The program can gather system statistics of execution of the process under the measurement. From the algorithm performance measurement perspective, the following statistics gathered by the program are important:

- user time – Total number of CPU-seconds that the process spent in user mode. Time spent on computation summed for every core.
- system time – Total number of CPU-seconds that the process spent in kernel mode. Time spent on computation summed for every core.

- elapsed wall clock time – Elapsed real-time as measured from the start until the stop of the program. The value can be lower than the sum of user time and system time, as the later values are a total sum of computation on every core.
- maximum resident set size – The memory allocated at peak for the process.
- major (requiring I/O) page faults – the number of major page faults that the process generated. A major page fault is triggered when Operating System moved a memory page into a disk and needs to fetch it from the disk. The value different from zero suggests that either system is heavily used or process requested more memory than could be provided by main memory.
- minor (reclaiming a frame) page faults – the number of minor page faults that the process generated. A minor page fault is triggered when Operating System claimed a memory page, but that memory page is still in the main memory in one of the system buffers and can be reclaimed by the process without flushing to nor reading from disk.
- voluntary context switches – the number of thread context switches made voluntarily by the process. The process gives back the CPU as it waits for some resource.
- involuntary context switches – the number of thread context switches forced by Operating System. The process is requested by the Operating System to pause its computations in order to share the CPU with other processes.
- file system inputs – the number of 512 bytes blocks read from the file system.
- file system outputs – the number of 512 bytes blocks written to file system.

The example output of GNU time testing graph generation for test model with $y = 16$ is presented below. The elapsed (wall clock) time is a biased estimator of the required execution time of the program, as the process is preempted by the Operating System, and then it does not perform computations. On the other hand, user-time plus system-time is more accurate, but it sums data from both cores. The algorithm is not parallel, but the garbage collector task runs in a separate thread. Therefore, the cores are not equally loaded, and user time does not represent a correct value either. The time program was used to verify the correctness of the algorithm and measure the memory consumption.

```
Command being timed: "stack --work-dir .stack-work run --
--modelname=M1048K --storagetype=FILE --statesfile=states.txt
--transitionsfile=transitions.txt --serialization=STORE"
User time (seconds): 48.59
System time (seconds): 9.70
Percent of CPU this job got: 181%
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:32.10
Average shared text size (kbytes): 0
Average unshared data size (kbytes): 0
Average stack size (kbytes): 0
Average total size (kbytes): 0
Maximum resident set size (kbytes): 309472
Average resident set size (kbytes): 0
Major (requiring I/O) page faults: 0
Minor (reclaiming a frame) page faults: 133106
Voluntary context switches: 75537
Involuntary context switches: 5436919
Swaps: 0
File system inputs: 0
File system outputs: 499712
Socket messages sent: 0
Socket messages received: 0
Signals delivered: 0
Page size (bytes): 4096
Exit status: 0
```

Another useful tool for troubleshooting the performance of the algorithm is Haskell GHC Profiler. Running the program under profiler allows gathering statistics of computation time and total memory allocation per line of the code. Therefore, it allows for finding places where the algorithm spends the most time or requires more memory allocations. Those methods or data structures that require the most time to compute or consume the most memory can be a focal point for further optimisations.

6.4 Memory leak

The tools mentioned in the previous sections helped to find issues with the implementation of the algorithm. One of the initial implementations of the algorithm consumed the same amount of memory while running version of the algorithm with all in-memory data structures as the version of the algorithm that was supposed to store only hashes of the states in-memory, while states and transitions were stored into a hard drive. This was odd as the latter version should have significantly smaller memory footprint. The state representation in the tested models requires ten times more memory than its hash representation.

The test was performed for the medium size model with the y = 16 that created roughly a million system states. With the help of the GNU *time* program, the following problem with the algorithm was spotted. The time required to perform computation on the CPU was at maximum 4094 s (which is 68 min 23 s)

– assuming no parallelisation and adding user time to system time. Due to poor data locality in memory, the program produced almost 8 million major page faults requiring the OS to fetch the missing page from a hard disk (swap area), resulting in the process's voluntary context switch. This problem extended total execution time to 16 hours 43 min 35 seconds. The output of the GNU time program is presented below in the tab. 6.2. The performance of the algorithm with the bug present is presented in the middle column, while performance of the algorithm after fixing the bug, in the right column.

The test was performed on the weaker machine with only 8 GB of main memory, therefore the previous version that required 16 GB of memory produced eight million page faults. File system IO was, as well, more significant and two-directional in the buggy example. This was related to storing and fetching the pages to and from a swap area on a hard disk.

Table 6.2: Number of states and transtions depending on the value of parameter y for the test model.

gnu statistic name	program with bug	after fixing bug
Command being timed	cabal run	stack –work-dir .stack-work run
User time (seconds)	3428.50	48.59
System time (seconds)	665.94	9.70
Percent of CPU this job got	6%	181%
Elapsed (wall clock) time (h:mm:ss)	16:43:35	0:00:32.10
Maximum resident set size (kbytes)	16010428	309472
Major (requiring I/O) page faults	7980961	0
Minor (reclaiming a frame) page faults	10085102	133106
Voluntary context switches	7990295	75537
Involuntary context switches	278807	5436919
File system inputs	69585408	0
File system outputs	25317432	499712

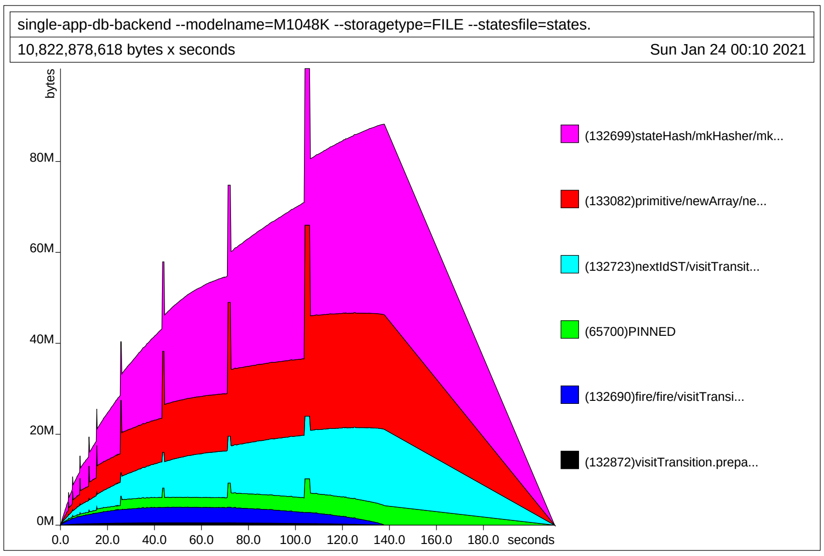
Thanks to the Haskell GHC Profiler, it was found out that there is a problem with allocation of the memory in the low-level buffers used primarily by the **ByteString** library. The memory was constantly allocated, and nothing was freed. The problem stemmed from the way how **ByteString** library optimises the allocation of the memory. The library request a 4 kB wide buffer and all newly created byte strings are stored in that buffer one after another. The garbage collector is allowed to free the buffer if all strings stored in it are no longer needed. The algorithm, however, interleaved the creation of the string representation of the states with the calculation of the hash of the state. So, the buffer allocated by the library had alternated a bigger string representation of the system state with a smaller string of hash of the state. The pointer to the hash was kept in memory in the dictionary, so even though string representation of the states were no longer needed, the garbage collector could not free any of the buffers. In that case, the main memory was fragmented, and the library did not clean it. That is why there was no difference in maximum resident memory size between a version storing everything in memory and a version that stored only hashes. On the development page of the **ByteString** library, there is an open ticket regarding this issue [73] since 2018. The solution that allowed the garbage collector to free the buffers was performing an additional copy operation of the hashes into **ShortByteString**, that allocates all the necessary memory on demand. So, additional memory copy allowed freeing more than 15.5 GB of memory. The number of file system inputs dropped to 0 together with the number of major page faults. The computation time, therefore, decreased from 16 hours to just 32 seconds.

Fig. 6.4 presents results of another optimisation of memory consumption as an output of Haskell GHC Profiler. The hashes of the states are stored in the dictionary, which is implemented as a hash array on top of a dynamically resizeable vector. Vector is a wrapper for the standard array, but it is able to resize itself on demand. If a new element is added to the vector while there is no more space for it, the algorithm allocates another array twice as big, then copies all elements from a smaller array to the new one, and adds the new element then disposes of the old array. If the element from a vector is removed and the number of occupied slots is lower than a quarter of all, then the twice smaller array is allocated and all elements are copied into a new array, while the bigger one is disposed of. The vector works well in the average case of multiple addition and removal of elements. However, the algorithm for graph generation only adds new elements and never removes them. This pattern of doubling the memory, copying and later disposal of the smaller array can be seen in the upper Figure as a comb-like structure in memory allocated in time. If the number of states could be estimated a priori than one allocation for the sufficient memory could be done, therefore removing the necessity for additional memory copies and allocations. The bottom Figure presents the version of the algorithm that performs a single allocation. That optimisation allowed to decrease the execution time by 20% and maximum required memory by 20%.

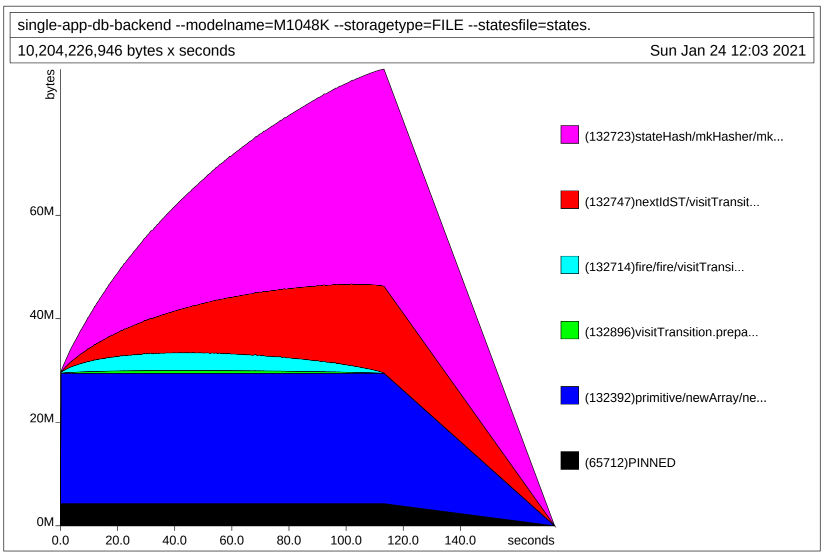
By investigation of the profiler results from Fig. 6.4, it is evident that the library uses so-called Boxed types, i.e. the hash array stores pointers to data rather than actual data. In Figure, there are three different stripes marked as: stateHash/mkHasher – calculated hash of the state, nextIdST – calculated index of the state, primitive/newArray – the memory required for the array, that contains pointers to the two previous sets of data. Further optimisation could use Unboxed types and store raw data within the array, removing all additional pointers. However, the current implementation of the dictionary library does not allow usage of Unboxed types.

6.5 Algorithm performance

The theoretical time and memory complexity of the algorithm generating LTS graph was calculated in the previous chapter. Here, the confirmation of the theoretical estimations is measured on test machine. The



(a) Small initial allocation of memory and further resizing



(b) Single initial allocation of enough memory to store all the elements

Figure 6.4: Memory profiling results (M1048K y=16)

program to calculate the LTS graph was compiled with multithreaded options, which does not speed some the algorithm, but it allows running garbage collection in separate thread. The following measurements were performed:

- a relative computation time of the algorithm for the different data structure set on a small model.
- a memory consumption of the algorithm with a fixed well performing data structures for different value of the meta parameter y.
- a computation time of the algorithm with a fixed well performing data structures for different value of the meta parameter y.

Firstly, the relative performance of the graph generation algorithm depending on the configuration was measured with the help of the criterion package [140]. The following aspects of the algorithm setup was taken into considerations:

- system state serialisation algorithm: serialisation to JSON format, serialisation to binary format by store library.
- system state hashing algorithm: sha256, city, custom.
- in memory data structure for storing hashes: basic hashtable io, basic hash table st, bitset.
- storing system states and transitions during computations: keeping in-memory, to plain disk file, to SQLite database.
- cache size when storing to database.

Fig 6.5 presents the results prepared by the criterion package. For testing purposes the value of the meta-parameter $y = 5$ was taken into consideration. This value of parameter results in a model with exactly 10,000 states 6.2. The model was chosen to be small as the criterion package runs every test set up in at least ten runs. In the chart (Fig 6.5) the following setup are presented:

- INMEMORYIO – state hashes are stored in the basic hash table available in IO context.
- INMEMORY – state hashes are stored in the basic hash table available in ST context.
- FILE-BITSET – state hashes are stored in the bitset, this setup requires custom hashing.
- FILE-SERIALIZATION-ALGORITHM/JSON – serialisation algorithm which stores system state in a human-readable JSON format, the states are stored in disk file.
- FILE-SERIALIZATION-ALGORITHM/STORE – serialisation algorithm which stores system state in a proprietary compact binary format, the states are stored in disk file.
- FILE-HASHING-ALGORITHM/SHA256 – general purpose cryptographic hash which generates 32 byte long hash, data stored in file, binary serialisation.
- FILE-HASHING-ALGORITHM/CITY – general purpose non-cryptographic hash which generates 16 byte long hash, data stored in file, binary serialisation.
- FILE-HASHING-ALGORITHM/CUSTOM – specially designed hash for the test model which gen-

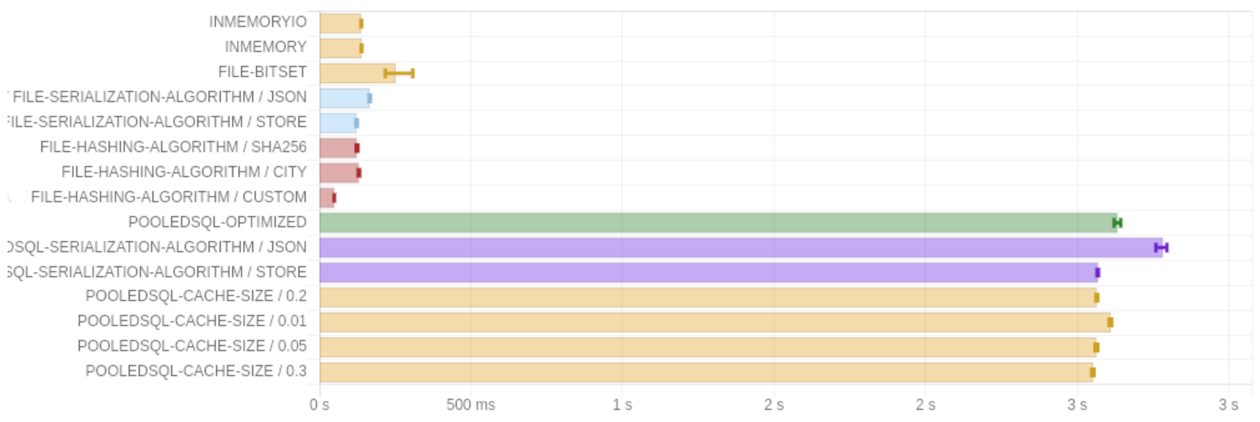


Figure 6.5: Different strategies impact on graph generation time

erates 4 byte long hash is guaranteed to be unique, data stored in file, binary serialisation.

- POOLEDSQL-OPTIMIZED – data stored in database, database configuration changes.
- POOLEDSQL-SERIALIZATION-ALGORITHM/JSON – serialisation algorithm which stores system state in a human-readable JSON format, the states are stored in database.
- POOLEDSQL-SERIALIZATION-ALGORITHM/STORE – serialisation algorithm which stores system state in a proprietary compact binary format, the states are stored in database.
- POOLEDSQL-CACHE-SIZE/0.2 – cache size of 20% of total state space size, data stored in database, binary serialisation.
- POOLEDSQL-CACHE-SIZE/0.01 – cache size of 1% of total state space size, data stored in database, binary serialisation.
- POOLEDSQL-CACHE-SIZE/0.05 – cache size of 5% of total state space size, data stored in database, binary serialisation.
- POOLEDSQL-CACHE-SIZE/0.3 – cache size of 30% of total state space size, data stored in database, binary serialisation.

Memory required to store all transitions and states in disk depends on the serialisation algorithm used. For the model with 10000 states: $y = 5$, binary format from the library store requires 1400000 bytes to keep all the states and 1040000 bytes to keep all transitions. The number of bytes required to store a single state and a single transition is constant and does not vary with the value of y. Therefore, for a single state, it requires 140 bytes and for a single transition 26 bytes (there are four times more transitions than states). On the other hand, JSON format requires 758890 bytes to keep all the states and 1871120 bytes to keep all transitions. The number of bytes required to store a single state and a single transition is not constant, and additionally varies with the value of y. For $y = 5$, a single state requires form 73 up to 77 bytes, and a single transition requires from 41 up to 48 bytes. For $y = 16$, a single state requires form 77 up to 87 bytes, and a single transition requires from 41 up to 53 bytes. This is because the numbers are encoded with varying number of characters. An example of JSON encoded state and transition are presented below:

```
[0,
[["X",1,[],[0,16]],
["X",1,[],[0,16]],
["X",1,[],[0,16]],
["X",1,[],[0,16]]]]

[0,
{"tag":"TLoop","contents":["A1",1]},
1]
```

JSON representation of the state is more compact than store representation, but the transition representation requires more memory than store representation. In total for $y = 5$, JSON requires 2.5 MB of disk space, while store requires 2.33 MB, additionally graph generation algorithm with store serialisation is able to complete computation faster in both cases while storing into disk file and into database.

Another dimension of the graph generation algorithm setup is the type of hashing algorithm used for hashing the state. For models up to 252 million of states, the size of the hash is constant and does not depend on the value of the meta-parameter. However, bigger state space would require adjustments in the custom hash method, as it is able to accommodate only 6-bit values of variable x. General purpose hashes may handle bigger state spaces without modifications, but required more memory to guarantee low probability of hash collisions. The results for different hashes show that sha256 and city hashes have similar performance, while the custom hash function allows the graph generation algorithm to complete computations twice faster. This trend is still visible for $y = 16$, and $y = 30$ with sha256 being slightly faster than city and custom hash being twice faster. The results of measurements with GNU time program of wall clock time are presented in the Table 6.3

Table 6.3: Computation time [min:sec] depending on the hash function for three values of the meta-parameter y.

hash	$y = 5$	$y = 16$	$y = 30$
sha256	0:00.123	0:19.71	5:05.05
city	0:00.130	0:20.25	4:32.77
custom	0:00.54	0:08.55	1:58.93

Next, the comparison of the storage containers for system states and transitions is analysed. In this case, both computation time and main memory consumption was measured with the GNU time. Four different setups were tested. First setup *in-memory* stores all transitions, system states, and hashes in main memory, second *bitset* stores transitions, and system states in disk file, but hashes are stored in bit set in main memory. Third setup *disk file* stores transitions, and system states in disk file, but hashes are stored in dictionary. Last setup *SQL database* stores all transitions, system states, and hashes in database on the disk. For last setup, one percent of the states are kept in in-memory cache. The computation time depending on the setup and the value of meta-parameter y is presented in Table 6.4. The main memory required by the algorithm at peak depending on the setup and the value of meta-parameter y is presented in Table 6.5. One can verify that

Table 6.4: Computation time [min:sec] depending on the storage container for states and transitions for three values of the meta-parameter y.

hash	$y = 5$	$y = 16$	$y = 30$
in-memory	0:00.55	0:10.61	3:38.85
bitset	0:00.83	0:06.81	1:14.98
disk file	0:00.54	0:08.55	1:58.93
sql database	0:02.48	3:39.17	44:26.96

the most memory is required by the *in-memory* computations, approximately 634 bytes per single system state. For *in-memory* computations and $y = 30$ it was possible to generate 59178 states per second. The least memory is required by the *sql database*, approximately 75 bytes per single system state, at the cost of longer computation time. For *sql database* and $y = 30$ it was possible to generate 4860 states per second. The fastest computation completion was achieved by *disk-file* and *bitset* setups. For $y = 30$, it was possible to generate 109000 and 175000 states per second respectively. *disk-file* storage setup required approximately 235 bytes per single system state. The *bitset* storage behaves differently. It is allocated four gigabytes array to store bit flags for all possible values of custom hash. The value of custom hash is an index in that array. Therefore, the memory required by the algorithm does not change significantly with the value of y. Additionally, to the visited states and transitions, there is maintained a queue of discovered yet unprocessed states which does not exceed two percent of all the states. Currently, in Haskell, a vector of boolean values requires a byte per element. The memory required by the bit set therefore could be decreased eight times if the flag was represented by a single bit.

The last group of tests explored the impact of the cache size on the computation speed with *SQL database* backend. In that case, no significant improvement can be seen.

For further analysis, the following setup was chosen:

- serialisation algorithm – store,
- hashing algorithm – custom,
- storage containers: *disk-file* and *bitset*.

Table 6.5: Maximum resident memory [kbytes] depending on the storage container for states and transitions for three values of the meta-parameter y.

hash	$y = 5$	$y = 16$	$y = 30$
in-memory	79.900	742.444	8.211.912
bitset	4.225.964	4.387.048	5.358.716
disk file	80.908	316.388	3.047.132
sql database	77.408	75.916	199.856

In Table 6.6, the results of GNU time program measurements of variant with storage container *disk-file* are presented. The results are limited to elapsed time and maximum resident memory size in kilobytes. Addi-

Table 6.6: Time and memory measurement depending on the model parameter y.

y	number of states	elapsed time [h:mm:ss.ms]	max. resident memory size [kB]	y	number of states	elapsed time [h:mm:ss.ms]	max. resident memory size [kB]
5	10000	00:00.55	76,756	14	614656	00:07.21	182,088
6	20736	00:00.62	73,480	15	810000	00:10.63	247,304
7	38416	00:00.77	76,752	16	1048576	00:13.98	312,304
8	65536	00:00.97	73,152	20	2560000	00:34.32	590,984
9	104976	00:01.39	73,144	25	6250000	02:35.65	1,625,704
10	160000	00:02.38	76,752	30	12960000	07:41.53	3,347,884
11	234256	00:02.95	84,076	40	40960000	41:14.79	11,917,644
12	331776	00:03.72	113,064	50	100000000	01:41:29.00	25,572,508
13	456976	00:04.54	132,376				

tionally, the results are visualised in Fig. 6.6 and Fig. 6.7. The previous graph presents the maximum resident memory size allocated for the program while computing the graph. In the graph, one can see that Haskell runtime environment requires approximately 80 MB memory. Therefore, for small state space sizes of up to 160.000 $y = 10$, the memory required by the algorithm is negligible and the memory reading is dominated by the requirements of the runtime environment. The maximum memory size is therefore approximately constant. The accuracy of maximum memory reading is biased and not accurate. Discrepancies of 10% were observed. However, starting from $y = 11$, the memory required for graph computation grows linearly with the size of the state space. Additionally, to the measured values, the function f with the given formulae was used:

$$f(n) = \begin{cases} 0.25n & \text{for } n > 160.000, \\ 80000 & \text{otherwise} \end{cases}$$

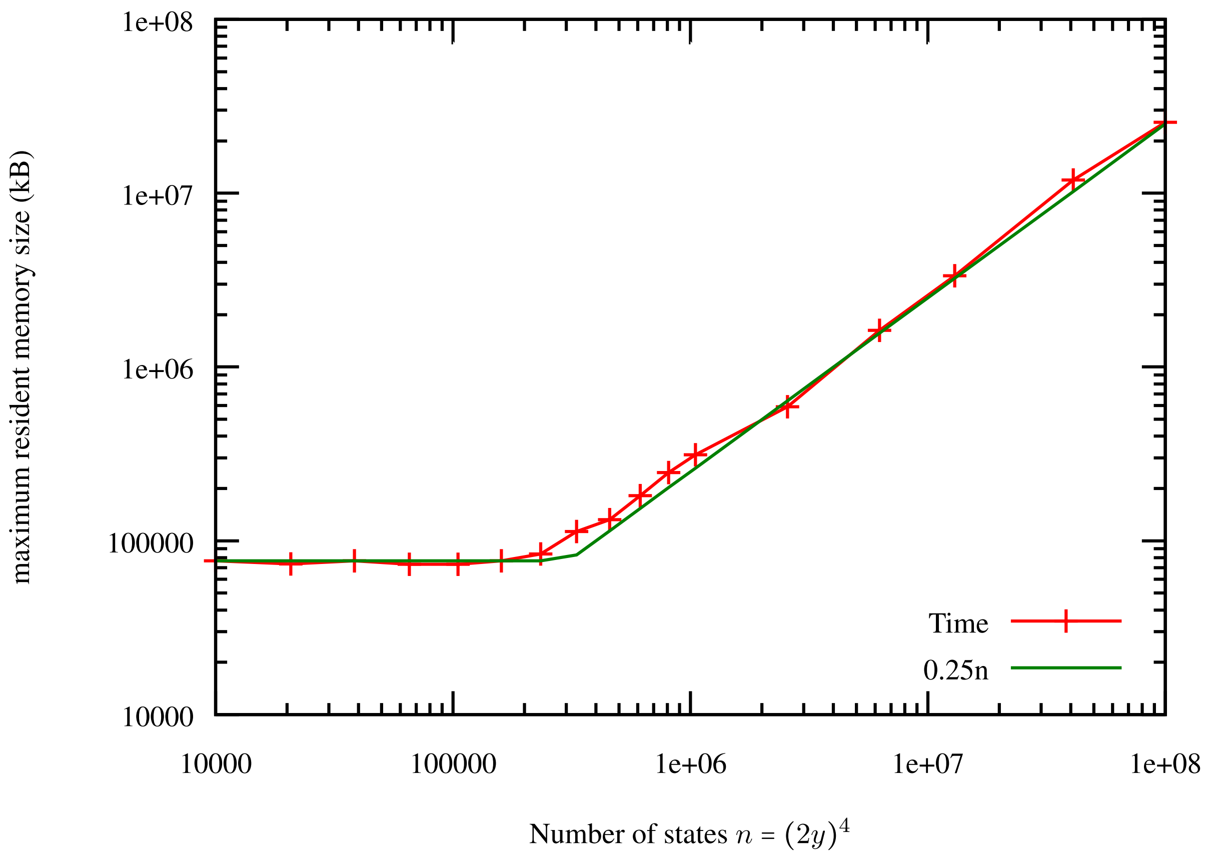


Figure 6.6: Maximum resident memory size in kilo bytes by GNU time

Just for comparison the data of time required to finish computation is presented in Fig. 6.7. The wall clock measurement are not precise and are highly influenced by the current load on the testing machine.

In Table 6.7, graph generation algorithm performance results with storage container *bitset* are presented. The measurements were done with criterion package. The package allows measuring only the computation time, but provides estimation of the measurement errors. The results were presented in the form of graph in Fig. 6.8. In addition to the measurement points, the following function was depicted in the chart:

$$f(n) = \begin{cases} 0.0035n & \text{for } n > 65.000, \\ 160 & \text{otherwise} \end{cases}$$

Extrapolating those result to the case of 10^9 states, it would require 3500s of computation which is slightly less than an hour. In order to do that, it would require adjusting the custom hash function to be able to

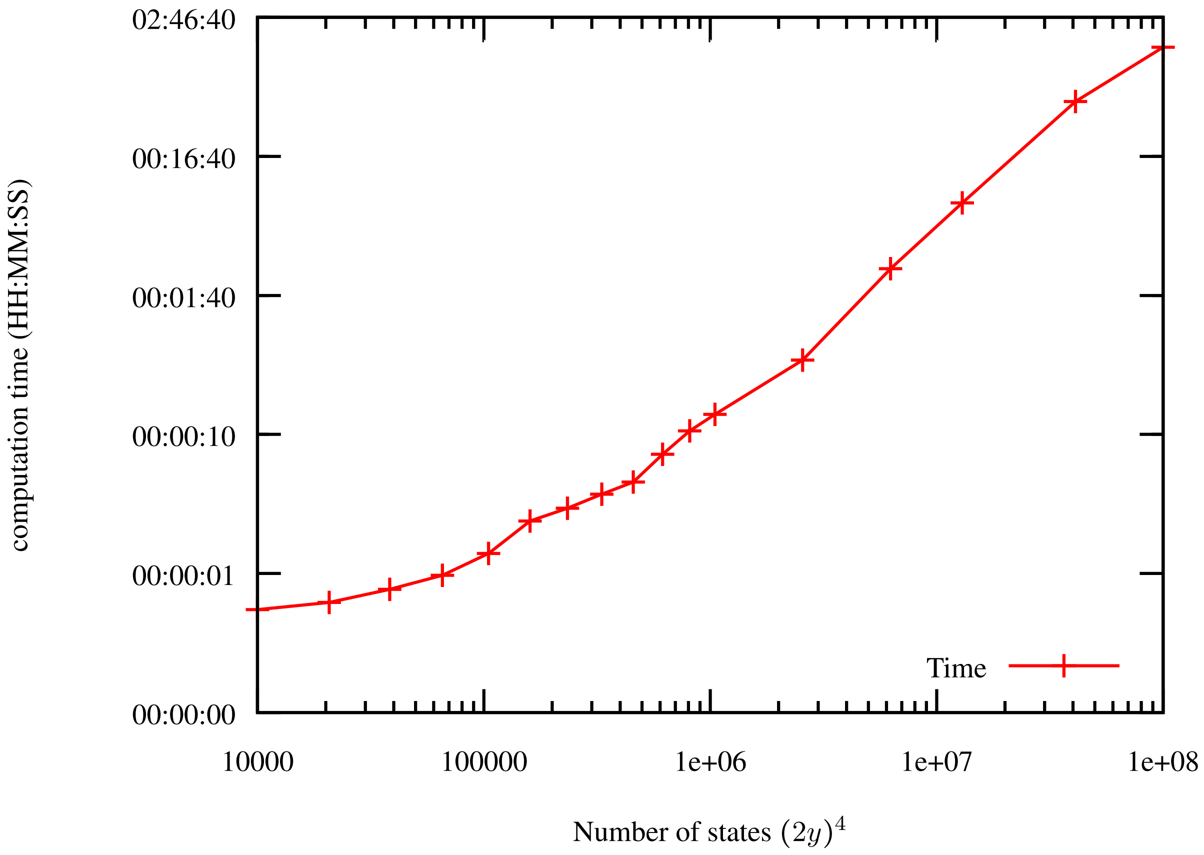


Figure 6.7: Elapsed (wall clock) time by GNU time

contain 7 bit value number. As now, maximum value of supported number of states by custom hash is for $y = 63$ which is 252 million states. 63 is the biggest 6 bit number.

6.6 Comparison with previous results

The results of the algorithm performance can be compared with the performance of the initial version before any optimisation were introduced. The same test model was checked for several values of meta parameter y. The results are gathered in tab. 6.8. Only the smaller values of meta-parameter were explored due to the significantly lower performance of the algorithm before optimisations were introduced. From collected data, we are able to compare models for a few medium-ranged values of meta parameters: $y = 10$ and $y = 14$. For $y = 10$ algorithm required 4 hours 54 min to compute LTS graph of 160 thousand states and at the peak allocated 3 GB of main memory. The generic version of the optimised version of the algorithm required 2 seconds 38 milliseconds of computation time and 77 MB of main memory. Even more optimised version of the algorithm taking advantage of highly optimised state encoding and bit set required only 0.5 s to complete the computation. The bit set allocated for the solution required 4 GB of main memory, but

Table 6.7: Alvis (bit-state) state space generation time depending on the model parameter y.

y	number of states	estimated computation time [ss.ms]	y	number of states	estimated computation time [m:ss.ms]
1	16	0.163	12	331,776	1.32
2	256	0.145	13	456,976	1.79
3	1,296	0.096	14	614,656	2.72
4	4,096	0.142	15	810,000	3.04
5	10,000	0.157	16	1,048,576	3.93
6	20,736	0.215	20	2,560,000	9.4
7	38,416	0.320	25	6,250,000	24.3
8	65,536	0.413	30	12,960,000	47.2
9	104,976	0.463	40	40,960,000	2:25.0
10	160,000	0.521	50	100,000,000	6:50.0
11	234,256	0.748			

Table 6.8: Alvis exploration algorithm before optimisation applied. Time and memory performance depending on the model parameter y.

y	number of states	elapsed time [h:mm:ss.ms]	max. resident memory size [kB]	y	number of states	elapsed time [h:mm:ss.ms]	max. resident memory size [kB]
5	10000	1:14.14	345,640	10	160000	4:54:55	3,040,780
6	20736	4:58.67	493,840	11	234256	10:47:13	4,217,660
7	38416	17:32.26	735,288	12	331776	22:24:54	6,879,968
8	65536	50:58.07	1,265,068	13	456976	41:03:46	8,222,540
9	104976	2:13:13	2,173,508	14	614656	73:42:49	11,813,432

its size does not depend on the size of the problem. So, the generic algorithm achieved almost 7500 times acceleration, while optimised packing of the state achieved further 35000 times acceleration. For $y = 14$ algorithm required 73 hours 43 min to compute LTS graph of 614 thousand states and at the peak allocated 12 GB of main memory. The generic version of the optimised version of the algorithm required 7.2 seconds of computation time and 182 MB of main memory. Optimised version of the algorithm using bit set required only 2.7 s to complete the computation. The bit set allocated for the solution again required 4 GB of main memory, but its size does not depend on the size of the problem. The generic algorithm achieved almost

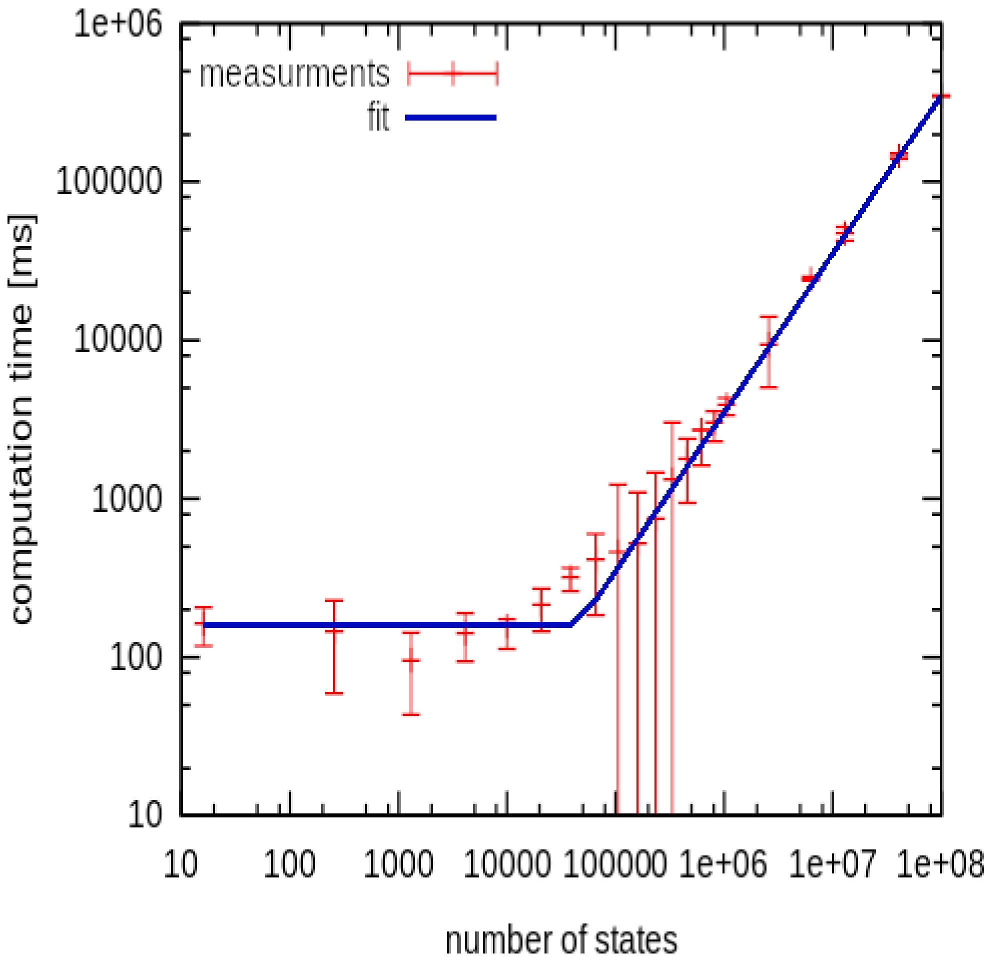
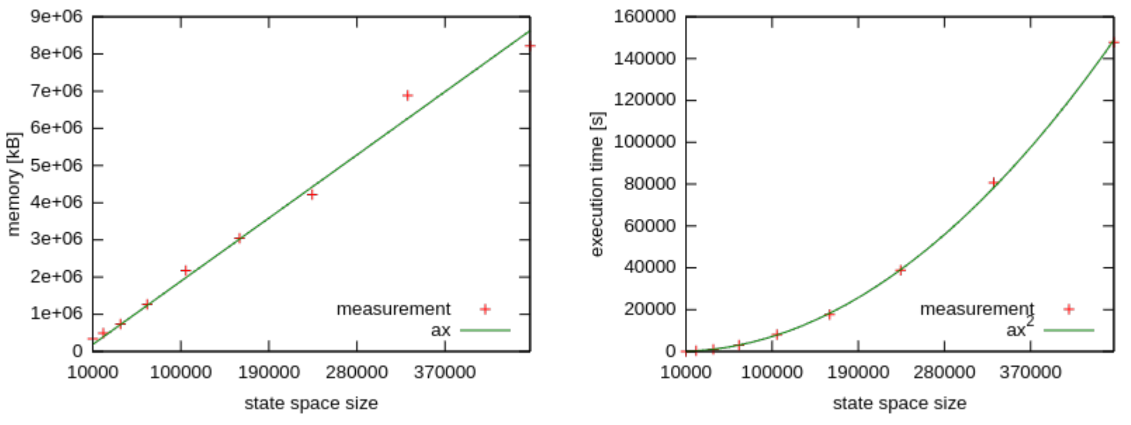


Figure 6.8: Computation time measurment depending on the size of the model.

37000 times acceleration, while optimised packing of the state achieved almost 100000 times acceleration. The resulting acceleration increases for the models with more states, due to the computation complexity differences between algorithms. The measured execution time data were fitted with quadratic model $t = a_t n^2$, where t is the execution time in seconds, $n = (2y)^4$ is the number of states, and a_t is an unknown parameter which can be estimated by fitting the model to measured data. Fitting procedure resulted with an estimation for the parameter $a_t = 7.129e - 07 + / - 3.8e - 09(0.52\%)$. The measured maximum memory allocated by the generator were fitted with linear model $m = a_m n$, where m is the maximum allocated main memory in kB, $n = (2y)^4$ is the number of states, and a_m is an unknown parameter which can be estimated by fitting the model to measured data. The fitting procedure resulted with an estimation for the parameter $a_m = 18.88 + / - 0.44(2.354\%)$.

Extrapolation measurements using fitted models let us estimate how many resources are needed for bigger models. Tackling the model $y = 50$ would require 2 TB of main memory and roughly 220 years to complete, while $y = 18$ would require only 31.8 GB of main memory and 22 days 20 hours to complete and it is the biggest problem that could be addressed in the computer used for tests. Exhausting main memory would slow down the computations even further.



(a) Maximum memory size in kB **(b)** Elapsed (wall clock) time

Figure 6.9: Alvis LTS generation algorithm before optimisations performance (GNU time).

6.7 Comparison with Spin model checker

Let us provide short comparison of graph generation performance with the state of the art model checker. The model checker most similar to Alvis in principles is the Spin cf. Sec. 2.8. Both Alvis and Spin are explicit state model checkers, and both generate programs that represent intermediate and implicit representation of state space. Running both programs generates an explicit representation of the graph. Spin allows on-the-fly model checking and provides various optimisations like probabilistic model checking, but here we will focus only on the time required to generate of the explicit state space.

The equivalent Spin model of four threads incrementing a value in an infinite loop, firstly introduced in Sec. 6.2 is presented below. The model consists of three global variables. First two of them `values` and `pcs` are tables that correspond to explicit `value` of local variable in Alvis model and implicit program counter that each agent has in Alvis. Next variable `y` represents the constant which is defined explicitly in Alvis model, and is the meta parameter of the system model. Later on, the definition of agents having the same behaviour. In Spin model, a built-in variable $_{pid}$ is used to distinguish the processes from each other and to manipulate the state of the processes independently. The major difference between the languages is that in Alvis all variables are local to the agent and cannot be changed by another agent accidentally, while in Spin any process can update an arbitrary global variable and the designer has to be careful to update the state of the system correctly. Later instructions in Spin model describe the process. The first two statements after assertion are responsible for initialisation of the global variables. Next statement delimited by *do* and *od* keywords represents the concurrent instructions which are executed in an infinite loop. The instructions within Spin loop are conditionally executed. The conditions test program counter value of the process. First statement within Spin loop corresponds to Alvis `loop`, and second statement corresponds to Alvis `exec` instruction. The later statement atomically changes the value of the program counter and internal variable.

```
byte values[4];
byte pcs[4];
byte y = 30;

active [4] proctype agent_a()
{
   assert(_pid == 0 || _pid == 1 || _pid == 2 || _pid == 3);
   values[_pid] = 0
   pcs[_pid] = 0
   do
   :: (pcs[_pid] == 0) -> pcs[_pid] = 1;
   :: (pcs[_pid] == 1) -> atomic {
            pcs[_pid] = 0;
            values[_pid] = (values[_pid]+1) % y;
   }
   od
}
```

Compilation of the model and its execution was done by the following commands.

```
spin -a agent_a.pml
gcc -E -DNOREDUCE pan.c > ppan.c
gcc -O2 ppan.c -o agent_a
./agent_a
```

The flag switching of partial order reduction was set to have comparable results.

The runtime results depending on the various value of meta-parameter y are presented in tab. 6.9. Comparison of the results with the one gathered in tab. 6.7 for Alvis model and graph exploration algorithm with bit set optimisation presented there, we can see that the graph exploration of model $y = 50$ was faster than in the case of Spin model checker and model $y = 40$. Moreover, Spin failed to explore state space of the model with $y = 50$ with out of memory exception. However, the bit set optimisation is not a generic one and can be applied only to a limited class of models. Therefore, we should compare the results with the more generic setup of Alvis graph exploration algorithm that stores hashes of the state in main memory presented in tab. 6.6. In that case, Alvis graph exploration was slower for model y = 40: 41 mintues to 7 min in the case of Spin model checker. But we can see that memory requirements for Alvis were 12 GB, while for Spin they were almost 22 GB. The trend of Alvis being slower is kept for smaller models as well – model y = 30 7 min 41 s to 2 min 23 s in the case of Spin model checker. However, memory requirements were lower for Alvis generation algorithm for every tested model size. As a result, Alvis was still able to explore model y = 50 while Spin analysis result with out of memory exception. Spin generated an order of magnitude more states.

Table 6.9: Spin state space generation time depending on the model parameter y.

y	number of states	estimated computation time [s.ms]	memory [MB]	y	number of states	estimated computation [m:ss.ms]	memory [MB]
1	2401	0	128	12	4885751	2.33	352
2	14641	0.01	129	13	6502974	3.29	426
3	48516	0.02	131	14	8495218	4.4	518
4	119363	0.05	134	15	11414580	6.27	651
5	251015	0.11	145	16	13983873	7.8	769
6	452631	0.21	149	20	31461271	21.5	1,569
7	729306	0.33	162	25	81431914	1:04	4,353
8	1157383	0.57	182	30	1.54e+08	2:23	9,656
9	1734650	0.79	208	40	4.20e+08	7:02	21,815
10	2543938	1.18	245	50	OOM	11:30	OOM
11	3688903	1.97	298				

6.8 Profiling results

Haskell compiler GHC provides a time and space profiling system that allows precise measurement of the memory allocation and running time of particular methods. Profiling of application means compiling Haskell program with special flags enabled which slows down execution but allows tracking memory and execution time. An example excerpt from profiling shows how insertion into the hash map is decomposed by the profiler tab. 6.10.

Table 6.10: Profiling results

COST CENTRE	MODULE	% time	% alloc
lookup/go	Data.HashTable.ST.Basic	29.0	0.4
findSafeSlots.go	Data.HashTable.ST.Basic	10.0	0.1
visitTransition	AlvisModel.GraphGenNoTime	4.0	2.8
⋆>.\	Data.Store.Core	4.0	1.0
contramap.	Data.Store.Impl	3.5	6.6
gpoke	Data.Store.Impl	3.1	10.7
gpoke	Data.Store.Impl	2.9	2.9
gsize	Data.Store.Impl	2.5	4.8
primitive	Control.**Monad**.Primitive	2.4	0.3

In Listing 6.2, the results from profiler were added in the form of comments in the graph exploration main loop method. Profiling data consists of four data points per method. First two numbers refer to execution of that particular line of code, while later two numbers correspond to accumulated results from execution of that particular line and all invoked methods by that line. First number describes percentage of execution time, second describes percentage of memory allocated by the given line, third and fourth number describe accumulated execution time, and accumulated memory for all methods invoked by that line. From that, we can see that visitTransition methods consumed 97% of total execution time and required almost 97% of total memory allocated. Later on, we can see that this is evenly spread for three methods: searching new state if it was not yet visited, cf. line 4, insertion of new states reachable from current state into exploredService line 9, and insertion of transitions into exploredService line 15. From the analysis we can see that $fire$ method was responsible for just a tiny fraction of resources, cf. line 2.

```
1  visitTransition toVisitService hasher exploredService idGenerator model (fromStateId, fromState) transition = do -- 4.0 2.8 97.0 96.6
2      let newStates = fire model transition fromState -- 0.2 0.0 1.0 2.1
3      let hashedStates = map (\s -> (stateHash hasher s, s)) newStates -- 0.5 2.9 2.2 7.8
4      lookedUpStates <- findAlreadyExploredStates exploredService hashedStates -- 0.6 1.0 33.0 8.2
5      let zippedStates = zip hashedStates lookedUpStates -- 0.2 1.2 0.2 1.2
6      let (alreadyStoredStates,toBeStoredStates) = partition (isAlreadyStored) zippedStates
7      let toBeStoredTransitions = map (prepareTransitionToStoreToAlreadyStoredState fromStateId transition) alreadyStoredStates -- 0.3
           ↪ 1.3 0.3 1.7
8      preparedToBeStored <- mapM (addIndexToHashedState) toBeStoredStates -- 0.3 0.4 0.4 0.4
9      ids <- addExploredStates exploredService preparedToBeStored -- 0.5 0.8 35.0 38.2
10
11     let toBeScheduledStates = zipWith (prepareStateToStore) toBeStoredStates ids -- 0.1 0.3 0.2 0.4
12     scheduleStatesToVisit toVisitService toBeScheduledStates -- 0.6 1.1 0.9 1.8
13
14     let toBeStoredTransitions2 = map (prepareTransitionToStoreFromNewStoredIds fromStateId transition) ids -- 0.1 0.7 0.2 0.8
```

```
addExploredTransitions exploredService (toBeStoredTransitions ++ toBeStoredTransitions2) -- 0.1 0.0 19.0 29.5

return ()
where
  ...
```

Listing 6.2: visitTransition

Chapter 7

Summary

Alvis formal modelling language was created to allow easy modelling and verification of formal models for software engineers. Therefore, Alvis language combines advantages of high-level programming languages with graphical language allowing modelling and visualisation of dependencies between model components in concurrent systems. At the foundation principles of Alvis, there was a capability of formal verification of designed model in an automated fashion.

At the time when the research was commenced, the Alvis language had a solid theoretical background. The rigorous mathematical foundation of Alvis Modelling language defined the syntax and semantics of the language. There was as well a tool supporting Alvis graphical layer modelling, namely Alvis IDE – a graphical user interface application that allows designing of the hierarchical Alvis models and syntax highlighting for the Alvis code layer. However, what was missing was the tools supporting processing and analysis of Alvis models according to the theoretical rules governing the semantics of the Alvis models. The most important missing tool was Alvis compiler.

Alvis compiler purpose was to validate syntax of Alvis model, report inconsistencies in the model if they were encountered, and allow efficient and automated generation of an Intermediate Haskell Representation from an input model. Alvis syntax validation incorporates from both its code layer syntax validation, and checking consistency between its graphical layer and code layer. Reporting inconsistencies and problems found during validation allows quick localisation of the problem. Alvis compiler is able to detect problems with agents missing its graphical or code representation, not defined ports or procedures, mismatched data types, and any problems with code syntax. Moreover, thanks to the carefully designed data structures and algorithms, Alvis compiler is able to generate IHR representation from arbitrary Alvis model after its representation was flattened. Flattening process is replacing the hierarchy in the model with an equivalent single-page model – flat model.

During the course of research conducted on Alvis compiler, there were suggested and successfully introduced few alternation of the Alvis instruction set. The suggested improvements and simplifications were:

non-blocking communication instructions, and requirement for passive procedure to complete with exit instruction. The first one simplified the complicated semantics of the original select instruction. The later one limited the semantics of passive agent procedure completion to just a single instruction and eliminated the complicated case of cascading procedures completions over a calling chain. Now every procedure has to finish with exit instruction, and therefore it is not possible that calling instruction of another procedure is the last one. Therefore, we avoid situation when one procedure end, triggers end of another procedure, and so on.

As the goal of efficient Alvis compiler was achieved, the focus of the research was shifted into generation of Labelled Transition System from Intermediate Haskell Representation of Alvis Model. As a result, the initial, naïve algorithm generating LTS was significantly improved. It enables the Alvis toolbox to explore models consisting of a billion of states on a contemporary, moderate PC station in just a few hours. For medium-sized model optimisation allowed exploration of the LTS to be 37,000 times faster than before any optimisations. Now Alvis compiler toolbox is comparable with well-known Spin model checker and significantly faster than nuXmv in terms of time spent and memory required to explore full state space of the modelled system.

All goals that were set for this dissertation were achieved successfully. The following items should be mentioned while considering the most impactful contributions that were completed as part of this dissertation:

- Design of a set of efficient algorithms for formal Alvis model transformation into its Intermediate Haskell Representation.
- Implementation of the Alvis model syntax checker.
- Implementation of the proposed algorithms in the form of the Alvis compiler – tool that automatically verifies syntactical correctness of an Alvis model design and is able to create an Intermediate Haskell Representation reflecting the designed model.
- Extension of Alvis language with non-blocking communication instructions.
- Design of efficient algorithms for automatic generation of Labelled Transition System from Intermediate Haskell Representation.
- Test model construction for evaluation purpose of LTS generation algorithms.
- Test environment implementation for running proposed test models and different LTS generation algorithms with a reliable performance measurements.
- Evaluation of the results.

Those tasks were successfully completed as part of the research conducted for this dissertation. The tasks' conclusion confirms completion of all goals as raised at the beginning of the dissertation. Therefore, the thesis statement was strongly confirmed. However, the dissertation does not deplete all the topics related to formal model verification with Alvis toolbox. During the course of conducted research, the following aspects were just signified and discovered but were not explored further. To name a few tasks, the following list could be compiled for future extension of Alvis toolbox capabilities:

- Integration of LTL verification algorithms with the Alvis compiler tools.
- Inclusion of the proof of concept implementation of algorithms explored by the dissertation with main Alvis project tool distribution.
- Exploration of partial order reduction methods for optimization of state space exploration.
- Other automatic state space reduction techniques considering hierarchical model definitions and leveraging the symmetries in the Alvis model.
- Further extension of Alvis Modelling Language with built-in subprocedures that could simplify more complex model design.
- Introduction of a system layer that allows to verify models in crush-recovery faults model.

Appendix A

Alvis examples

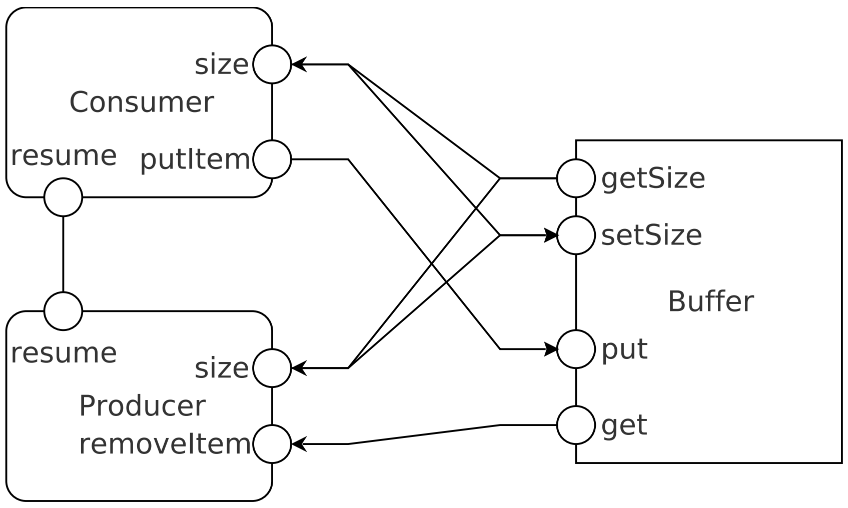
```
agent Buffer {
 value :: Int = 0;
 queue :: Seq Int = empty;
 MAX_SIZE :: Int = 3;

 proc (length queue > 0) get {
   value = index queue 0;
   queue = drop 1 queue;
   out get value;
   exit;
 }

 proc (length queue < MAX_SIZE) put {
   in put value;
   queue = value |> value;
   exit;
 }
}
```

Listing A.1: Blocking Bounded Buffer Alvis model

```
agent Producer {
 value :: Int = 0;
 queueSize :: Int = 0;
 loop {
   value = pick [1..3];
   in getSize queueSize;
   select {
     queueSize > 3: {
       loop {
         in resume;
   }}}
   out push value;
   queueSize = queueSize + 1;
```



```
    out setSize queueSize;
    select {
      queueSize == 1: {
        loop {
          out resume;
}}}}}
```

Listing A.2: Error prone Bounded Buffer Alvis model

```
agent Consumer {
  value :: Int = 0;
  queueSize :: Int = 0;
  loop {
    in getSize queueSize;
    select {
      queueSize == 0: {
        loop {
          in resume;
    }}}
    in removeItem value;
    queueSize = queueSize − 1;
    out setSize queueSize;
    select {
      queueSize == 2: {
        loop {
          out resume;
}}}}}
```

Listing A.3: Error prone Bounded Buffer Alvis model

```
agent Buffer {
  value :: Int = 0;
  size :: Int = 0;

```

```
 proc getSize {
  out getSize size;
  exit;
 }

 proc setSize {
  in setSize size;
  exit;
 }

 proc get {
  out get value;
  exit;
 }

 proc put {
  in put value;
  exit;
 }
}
```

Listing A.4: Error prone Bounded Buffer Alvis model

Bibliography

[1] L. Aceto, A. Ingófsdóttir, K. Larsen, and J. Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, Cambridge, UK, 2007.

[2] S. Adams. Functional pearls efficient sets—a balancing act. *Journal of Functional Programming*, 3(4):553–561, 1993.

[3] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181 – 185, 1985.

[4] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed computing*, 2(3):117–126, 1987.

[5] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2 – 34, 1993.

[6] Y. A. Ameur and K.-D. Schewe. Abstract State Machines, Alloy, B, TLA, VDM, and Z. In *4th International Conference, ABZ 2014. Proceedings*, volume 8477. Springer, 2014.

[7] A. Appleby. Murmurhash project, 2008.

[8] A. Appleby. Smhasher & murmurhash, 2012.

[9] E. A. Ashcroft. Proving assertions about parallel programs. *Journal of Computer and System Sciences*, 10(1):110–135, 1975.

[10] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, London, UK, 2008.

[11] K. Balicki and M. Szpyrka. Formal definition of XCCS modelling language. *Fundamenta Informaticae*, 93(1-3):1–15, 2009.

[12] J. Baniewicz. *METODY FORMALNEJ ANALIZY SYSTEMóW WBUDOWANYCH CZASU RZECZYWISTEGO*. PhD thesis, AGH University of Science and Technology, 2018.

[13] J. Baniewicz and M. Szpyrka. Formal verification of real-time systems developed for single-processor platform with alvis. In *26th International Conference Mixed Design of Integrated Circuits and System (MIXDES)*, page 129. Institute of Electrical and Electronics Engineers (IEEE), Toruń, Poland, June 27–29 2019.

[14] B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.-C. Filliâtre, E. Giménez, H. Herbelin, et al. The coq proof assistant reference manual. *INRIA, version*, 6(11), 1999.

[15] K. Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[16] G. Behrmann, A. David, and K. G. Larsen. *A Tutorial on Uppaal*, pages 200–236. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[17] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer–Verlag, Oct. 1995.

[18] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. *Lecture Notes on Concurrency and Petri Nets*, 3098, 2004.

[19] J. A. Bergstra, A. Ponse, and S. A. Smolka, editors. *Handbook of Process Algebra*. Elsevier Science, Upper Saddle River, NJ, USA, 2001.

[20] J. Biernacki. Alvis models of safety critical systems state-base verification with nuXmv. In *Proceedings of the Federated Conference on Computer Science and Information Systems*, pages 1701–1708, 2016.

[21] J. Biernacki. *APPLICATION OF FUNCTIONAL PARADIGM TO FORMAL ANALYSIS OF SYSTEMS MODELLED WITH ALVIS LANGUAGE*. PhD thesis, AGH University of Science and Technology, 2020.

[22] G. V. Bochmann. Finite state description of communication protocols. *Computer Networks (1976)*, 2(4):361 – 372, 1978.

[23] G. Brat, K. Havelund, S. Park, and W. Visser. Java pathfinder - second generation of a java model checker. In *In Proceedings of the Workshop on Advances in Verification*, 2000.

[24] E. Brinksma. Information processing systems, open systems interconnection LOTOS. Technical Report ISO 8807, ISO/IEC JTC 1/SC 7 Software and systems engineering, 1989.

[25] R. E. Bryant. Binary decision diagrams. In *Handbook of model checking*, pages 191–217. Springer, 2018.

[26] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142 – 170, 1992.

[27] A. Burns and A. Wellings. *Concurrent and real-time programming in Ada 2005*. Cambridge University Press, 2007.

[28] B. Campbell. Efficiency of purely functional programming, 2010.

[29] J. Carmo and A. Sernadas. Branching versus linear logics yet again. *Formal Aspects of Computing*, 2(1):24–59, 1990.

[30] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. The nuXmv symbolic model checker. In *Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*, pages 334–342. Springer, 2014.

[31] A. Ceccarelli and N. Silva. Qualitative comparison of aerospace standards: An objective approach. In *2013 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 331–336. IEEE, 2013.

[32] D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, C. McKinty, V. Powazny, F. Lang, W. Serwe, and G. Smeding. *Reference Manual of the LOTOS NT to LOTOS Translator (Version 5.4)*. INRIA/VASY, 2011.

[33] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. Verifying safety properties with the TLA+ proof system. In *International Joint Conference on Automated Reasoning*, pages 142–148. Springer, 2010.

[34] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV version 2: An opensource tool for symbolic model checking. In *Proceedings of International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, 2002. Springer-Verlag.

[35] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[36] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani. The MathSAT5 SMT Solver. In N. Piterman and S. Smolka, editors, *Proceedings of TACAS*, volume 7795 of *LNCS*. Springer, 2013.

[37] A. Civil. Aircraft accident report: controlled flight into terrain, american airlines flight 965, boeing 757-223, n651aa near cali, colombia, december 20, 1995. *Bogota: Aeronautica Civil*, 1996.

[38] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[39] E. M. Clarke. The birth of model checking. In *25 Years of Model Checking*, pages 1–26. Springer, 2008.

[40] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, Berlin, Heidelberg, 1982. Springer-Verlag.

[41] E. M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Formal methods in system design*, 9(1-2):77–104, 1996.

[42] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1512–1542, 1994.

[43] E. M. Clarke, O. Grumberg, M. Minea, and D. A. Peled. State space reduction using partial order techniques. *International Journal on Software Tools for Technology Transfer*, 2:279–287, 1999.

[44] J. Coens. Hotswapping haskell, 2017.

[45] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Computing Surveys (CSUR)*, 3(2):67–78, 1971.

[46] Y. Collet. xx hash, 2015.

[47] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[48] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[49] G. corp. Gatling project, stress tool, 2012.

[50] I. Corporation. Intel sse4 programming reference, 2007.

[51] P.-J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with “readers” and “writers”. *Communications of the ACM*, 14(10):667–668, 1971.

[52] S. Diehl. What I wish I knew when learning haskell, 2016.

[53] E. Dijkstra. Information streams sharing a finite buffer. *Information Processing Letters*, 1(5):179–180, 1972.

[54] E. W. Dijkstra. Cooperating sequential processes, technical report ewd-123, 1965.

[55] S. Donatelli and S. Haar. *Application and Theory of Petri Nets and Concurrency*. Springer, 2019.

[56] C. Done. Haskell performance: Dictionaries, 2017.

[57] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[58] C. Eisner and D. Peled. Comparing symbolic and explicit model checking of a software system. In D. Bošnački and S. Leue, editors, *Model Checking Software*, pages 230–239, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[59] E. A. EMERSON. Chapter 16 - temporal and modal logic. In J. VAN LEEUWEN, editor, *Formal Models and Semantics*, Handbook of Theoretical Computer Science, pages 995 – 1072. Elsevier, Amsterdam, 1990.

[60] E. A. Emerson. Model checking and the Mu-calculus. In N. Immerman and P. G. Kolaitis, editors, *Descriptive Complexity and Finite Models*, volume 31 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 185–214. American Mathematical Society, 1997.

[61] J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. Cadp a protocol validation and verification toolbox. In *International Conference on Computer Aided Verification*, pages 437–440. Springer, 1996.

[62] M. J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). In *International conference on fundamentals of computation theory*, pages 127–140. Springer, 1983.

[63] E. Freund. Ieee standard for system and software verification and validation (ieee std 1012-2012). *Software Quality Professional*, 15(1):43, 2012.

[64] A. Gakhov. *Probabilistic Data Structures and Algorithms for Big Data Applications*. BoD–Books on Demand, 2019.

[65] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[66] H. Garavel. Open/cæsar: An open software architecture for verification, simulation, and testing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 68–84. Springer, 1998.

[67] H. Garavel. XTL: A metalanguage and tool for temporal logic modelchecking. In *In Proceedings of the International Workshop on Software Tools for Technology Transfer STTT98 (Aalborg, Denmark)*, pages 33–42, 1998.

[68] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2006: A toolbox for the construction and analysis of distributed processes. In *Computer Aided Verification (CAV'2007)*, volume 4590 of *LNCS*, pages 158–163, Berlin, Germany, 2007. Springer.

[69] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2011: a toolbox for the construction and analysis of distributed processes. *International Journal on Software Tools for Technology Transfer (STTT)*, 15(2):89–107, 2013.

[70] P. Godefroid. Using partial orders to improve automatic verification methods. In *International Conference on Computer Aided Verification*, pages 176–185. Springer, 1990.

[71] Google. *Building Software Systems at Google and Lessons Learned*, Stanford Computer Science Department Distinguished Computer Scientist Lecture, 2010.

[72] Google. Farmhash, 2014.

[73] N. Hambüchen. Bytestring pinned memory can be leaky, 2017.

[74] V. Hanquez. Crypto.hash.sha256. Technical report, Industrial Haskell Group, 2016.

[75] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[76] W. Hillel. Learn TLA+, 2019.

[77] R. Hinze and R. Paterson. Finger trees: A simple general-purpose data structure. *J. Funct. Program.*, 16(2):197–217, march 2006.

[78] C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[79] C. A. R. Hoare and J. C. Shepherdson. *Mathematical Logic and Programming Languages*. Prentice Hall, 1985.

[80] A. Holdings. Arm architecture reference manual, armv8, for armv8-a architecture profile, 2019.

[81] V. Holub. Murmur hash 2.0.

[82] G. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, first edition, 2003.

[83] G. J. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997.

[84] G. J. Holzmann. *Explicit-State Model Checking*, pages 153–171. Springer International Publishing, Cham, 2018.

[85] H. Hubbeling. An introduction to modal logic (een herdruk met correcties van de eerste druk van 1968), 1974.

[86] J. Hughes. Programming with arrows. In *International School on Advanced Functional Programming*, pages 73–129. Springer, 2004.

[87] I. Intel. and ia-32 architectures software developers manual–volume 1: Basic architecture. *Intel Corporation*, 253665, 2008.

[88] I. Intel. and ia-32 architectures software developers manual–volume 2 (2a, 2b): Instruction set reference, az. *Intel Corporation*, 253666, 2008.

[89] C. N. Ip and D. L. Dill. Better verification through symmetry. *Formal methods in system design*, 9(1-2):41–75, 1996.

[90] B. Jenkins. Spooky, 2012.

[91] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, volume 1–3. Springer-Verlag, Berlin, Germany, 1992-97.

[92] K. Jensen and L. Kristensen. *Coloured Petri nets. Modelling and Validation of Concurrent Systems*. Springer, Heidelberg, 2009.

[93] I. Jones. The haskell cabal, a common architecture for building applications and libraries, 2005.

[94] M. Kleppmann. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. " O'Reilly Media, Inc.", 2017.

[95] D. E. Knuth. *The art of computer programming: Volume 3: Sorting and Searching*. Addison-Wesley Professional, 1998.

[96] D. Kozen. Results on the propositional μ-calculus. In M. Nielsen and E. M. Schmidt, editors, *Automata, Languages and Programming*, pages 348–359, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[97] W. Kumoń. Implementation of the distributed ltl model checking algorithm. Master's thesis, AGH University of Science and Technology, 2019.

[98] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[99] L. Lamport. Proving the correctness of multiprocess programs. *IEEE transactions on software engineering*, SE-3(2):125–143, 1977.

[100] L. Lamport. Specifying concurrent systems with TLA+. *Calculational System Design*, pages 183–247, April 1999.

[101] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, June 2002.

[102] L. Lamport. The PlusCal algorithm language. *Theoretical Aspects of Computing-ICTAC 2009, Martin Leucker and Carroll Morgan editors. Lecture Notes in Computer Science, number 5684, 36-60.*, January 2009.

[103] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. In *Concurrency: the Works of Leslie Lamport*, pages 203–226. ACM Books, 2019.

[104] L. Lamport and Y. Yu. TLC–the TLA+ model checker, 2001.

[105] F. Lang. Exp.open 2.0: A flexible tool integrating partial order, compositional, and on-the-fly verification methods. In *Proceedings of the 5th International Conference on Integrated Formal Methods*, IFM'05, pages 70–88, Berlin, Heidelberg, 2005. Springer-Verlag.

[106] L. LaPiana and F. Bauer. Mars climate orbiter mishap investigation board phase i report, 1999.

[107] D. Lea. *Concurrent programming in Java: design principles and patterns*. Addison-Wesley Professional, 2000.

[108] D. Lea, J. Bowbeer, D. Holmes, and S. AG. Jsr 166: Concurrency utilities. `http://jcp.org/en/jsr/detail`, 2004.

[109] D. Leijen. Data.set. Technical report, Industrial Haskell Group, 2002.

[110] K. R. M. Leino. Accessible software verification with dafny. *IEEE Software*, 34(06):94–97, nov 2017.

[111] K. R. M. Leino. Modeling concurrency in dafny. In *School on Engineering Trustworthy Software Systems*, pages 115–142. Springer, 2017.

[112] K. R. M. Leino. Dafny, 2019.

[113] R. Leino. Dafny: An automatic program verifier for functional correctness. In *16th International Conference, LPAR-16, Dakar, Senegal*, pages 348–370. Springer Berlin Heidelberg, April 2010.

[114] F. J. Lin, P. Chu, and M. T. Liu. Protocol verification using reachability analysis: the state space explosion problem and relief strategies. In *Proceedings of the ACM workshop on Frontiers in computer communications technology*, pages 126–135, 1987.

[115] J. L. Lions. Ariane 5 flight 501 failure: Report by the enquiry board, 1996.

[116] M. Lipovača. *Learn you a haskell for great good!: a beginner's guide*. no starch press, 2011.

[117] L. Logrippo, M. Faci, and M. Haj-Hussein. An introduction to LOTOS: learning by examples. *Computer networks and ISDN systems*, 23(5):325–342, 1992.

[118] J. R. Lorch, Y. Chen, M. Kapritsos, B. Parno, S. Qadeer, U. Sharma, J. R. Wilcox, and X. Zhao. Armada: Low-effort verification of high-performance concurrent programs. In *International Conference on Programming Language Design and Implementation (PLDI)*, pages 197–210. ACM, ACM, June 2020.

[119] J. Magee and J. Kramer. *Concurrency: State Models and Java Programs*. Wiley Publishing, 2nd edition, 2006.

[120] C. Marie. Data.digest.xxhash. Technical report, Industrial Haskell Group, 2017.

[121] S. Marlow. *Haskell 2010 Language Report*, 2010.

[122] A. Martin. Adequate sets of temporal connectives in ctl. *Electr. Notes Theor. Comput. Sci.*, 52:21–31, 01 2001.

[123] R. C. Martin. The dependency inversion principle. *C++ Report*, 8(6):61–66, 1996.

[124] R. C. Martin. Design principles and design patterns. *Object Mentor*, 1(34):597, 2000.

[125] R. C. Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.

[126] R. C. Martin. *Clean Code-Refactoring, Patterns, Testen und Techniken für sauberen Code: Deutsche Ausgabe*. MITP-Verlags GmbH & Co. KG, 2013.

[127] R. Mateescu and J. I. Requeno. On-the-fly model checking for extended action-based probabilistic operators. *International Journal on Software Tools for Technology Transfer*, 20(5):563–587, Oct 2018.

[128] R. Mateescu and M. Sighireanu. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. *Sci. Comput. Program.*, 46(3):255–281, Mar. 2003.

[129] P. Matyasik, M. Szpyrka, and M. Wypych. Generation of Java code from Alvis model. In *International Conference of Computational Methods in Sciences and Engineering (ICCMSE 2015)*, volume 1702 of *AIP Conference Proceedings*, pages 100013–1–100013–4, Athens, Greece, March 20-23 2015. AIP Publishing.

[130] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275, 1996.

[131] Microsoft. High-level TLA+ specifications for the five consistency levels offered by Azure Cosmos DB, 2018.

[132] R. Milner. *Communication and concurrency*, volume 84. Prentice hall New York etc., 1989.

[133] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The definition of standard ML: revised*. MIT press, 1997.

[134] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[135] Netflix. Chaos monkey, 2016.

[136] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How amazon web services uses formal methods. *Communications of the ACM*, 58:66–73, 03 2015.

[137] U. Norell, N. A. Danielsson, A. Abel, and J. Cockx. The agda wiki, 2020.

[138] G. Norman, D. Parker, and J. Sproston. Model checking for probabilistic timed automata. *Formal Methods in System Design*, 43(2):164–190, 2013.

[139] R. Osherove. *The Art of Unit Testing: With Examples in. Net*. Manning Publications Co., 2009.

[140] B. O'Sullivan. Criterion. Technical report, Industrial Haskell Group, 2019.

[141] B. O'Sullivan. Data.aeson. Technical report, Industrial Haskell Group, 2019.

[142] B. O'Sullivan, J. Goerzen, and D. Stewart. *Real World Haskell*. O'Reilly Media, Sebastopol, CA, USA, 2008.

[143] B. O'Sullivan, J. Goerzen, and D. B. Stewart. *Real world haskell: Code you can believe in*. " O'Reilly Media, Inc.", 2008.

[144] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta informatica*, 6(4):319–340, 1976.

[145] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):455–495, 1982.

[146] T. Peierls, B. Goetz, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes. *Java Concurrency in Practice: JAVA CONCURRENCY PRACT _p1*. Pearson Education, 2006.

[147] D. Peled. Combining partial order reductions with on-the-fly model-checking. In *International Conference on Computer Aided Verification*, pages 377–390. Springer, 1994.

[148] W. Penard and T. van Werkhoven. On the secure hash algorithm family. *Cryptography in Context*, pages 1–18, 2008.

[149] C. A. Petri. Communication with automata. Technical report, Supplement 1 to Technical Report RADC-TR-65-377, 1965.

[150] G. Pike. Farm hash, 2008.

[151] G. Pike and J. Alakuijala. City hash, 2011.

[152] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57. IEEE, 1977.

[153] A. N. Prior. *Past, present and future*, volume 154. Clarendon Press Oxford, 1967.

[154] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, UK, 1982. Springer-Verlag.

[155] L. Rierson. *Developing safety-critical software: a practical guide for aviation software and DO-178C compliance*. CRC Press, 2017.

[156] F. Rosa-Velardo. Coding mobile synchronizing petri nets into rewriting logic. *Electr. Notes Theor. Comput. Sci.*, 174:83–98, 04 2007.

[157] R. Singh. International standard iso/iec 12207 software life cycle processes. *Software Process Improvement and Practice*, 2(1):35–50, 1996.

[158] S. S. Skiena. *The Algorithm Design Manual*. Springer Publishing Company, Incorporated, 2nd edition, 2008.

[159] M. Sloan. Data.store. Technical report, Industrial Haskell Group, 2019.

[160] J. Stecklein, J. Dabney, B. Dick, B. Haskins, R. Lovell, and G. Moroney. Error cost escalation through the project life cycle. *Source of Acquisition NASA Johnson Space Center*, 2004.

[161] D. Stewart and C. Duncan. Data.bytestring. Technical report, Industrial Haskell Group, 2017.

[162] M. Szpyrka. Analysis of VME-Bus communication protocol – RTCP-net approach. *Real-Time Systems*, 35(1):91–108, 2007.

[163] M. Szpyrka. *Modelowanie systemów współbieżnych w języku Alvis*. Wydawnictwa AGH, 2013.

[164] M. Szpyrka and P. Matyasik. Formal modelling and verification of concurrent systems with XCCS. In *Proceedings of the 7th International Symposium on Parallel and Distributed Computing (ISPDC 2008)*, pages 454–458, Krakow, Poland, July 1-5 2008.

[165] M. Szpyrka, P. Matyasik, J. Biernacki, A. Biernacka, M. Wypych, and L. Kotulski. Hierarchical communication diagrams. *Computing and Informatics*, 35(1):55–83, 2016.

[166] M. Szpyrka, P. Matyasik, and R. Mrówka. Alvis – modelling language for concurrent systems. In P. Bouvry, H. Gonzalez-Velez, and J. Kołodziej, editors, *Intelligent Decision Systems in Large-Scale Distributed Environments*, volume 362 of *Studies in Computational Intelligence*, chapter 15, pages 315–341. Springer-Verlag, 2011.

[167] M. Szpyrka, P. Matyasik, R. Mrówka, and L. Kotulski. Formal description of Alvis language with α^0 system layer. *Fundamenta Informaticae*, 129(1-2):161–176, 2014.

[168] M. Szpyrka, P. Matyasik, L. Podolski, and M. Wypych. Simulation of multi-agent systems with Alvis Toolkit. In L. Rutkowski, M. Korytkowski, R. Scherer, R. Tadeusiewicz, L. Zadeh, and Z. J., editors, *Artificial Intelligence and Soft Computing. ICAISC 2017*, volume 10246 of *LNCS*, pages 599–608. Springer-Verlag, 2017.

[169] M. Szpyrka, P. Matyasik, and M. Wypych. Alvis language with time dependence. In *Proceedings of the Federated Conference on Computer Science and Information Systems*, pages 1607–1612, Krakow, Poland, 2013.

[170] M. Szpyrka, P. Matyasik, and M. Wypych. Generation of labelled transition systems for alvis models using haskell model representation. In *Proceedings of the 22nd International Workshop on Concurrency, Specification and Programming (CS&P 2013)*, volume 1032, pages 409–420, Warsaw, Poland, 2013. CEUR Workshop Proceedings.

[171] M. Szpyrka, P. Matyasik, M. Wypych, J. Biernacki, and L. Podolski. Alvis modelling language. Technical report, AGH-UST, 2016.

[172] M. Szpyrka, P. Matyasik, M. Wypych, J. Biernacki, and L. Podolski. *Alvis Modelling Language*, 2017.

[173] M. Szpyrka, L. Podolski, and M. Wypych. Modelling and verification of real-time systems with Alvis. In P. Kosiuczenko and L. Madeyski, editors, *Towards a Synergistic Combination of Research and Practice in Software Engineering KKIO 2017*, volume 733 of *Studies in Computational Intelligence*, pages 165–178. Springer-Verlag, 2018.

[174] M. Szpyrka, M. Wypych, J. Biernacki, and L. Podolski. Discrete-time systems modeling and verification with Alvis language and tools. *IEEE Access*, 6:78766–78779, 2018.

[175] G. Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*, 2002.

[176] S. Timnat and E. Petrank. A practical wait-free simulation for lock-free data structures. *ACM SIGPLAN Notices*, 49(8):357–368, 2014.

[177] P. Trentin. Spin: Introduction, 2018.

[178] R. Urban. Smhasher benchmark results, 2014.

[179] A. Valmari. A stubborn attack on state explosion. In *International Conference on Computer Aided Verification*, pages 156–165. Springer, 1990.

[180] A. Valmari. The state explosion problem. In *Advanced Course on Petri Nets*, pages 429–528. Springer, 1996.

[181] A. van der Ploeg. Data.sequence. Technical report, Industrial Haskell Group, 2015.

[182] M. Y. Vardi. Branching vs. linear time: Final showdown. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 1–22. Springer, 2001.

[183] R. Von Mises. *Selected papers of Richard von Mises*, volume 2. American Mathematical Society, 1964.

[184] M. Zocca. Haskell performance: Sequences, 2017.