An Algorithm and Implementation of Equivalence Checking in Pi-Calculus through Fresh-Register Automata

Jevgenijus Čistiakovas

A Dissertation

Presented to the University of Dublin, Trinity College

in partial fulfilment of the requirements for the degree of

Master in Computer Science

Supervisor: Dr. Vasileios Koutavas

April 2022

Declaration

I hereby declare that this dissertation is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at `http://www.tcd.ie/calendar`.

I have also completed the Online Tutorial on avoiding plagiarism 'Ready Steady Write', located at `http://tcd-ie.libguides.com/plagiarism/ready-steady-write`.

Jevgenijus Čistiakovas

2022-04-19

An Algorithm and Implementation of Equivalence Checking in Pi-Calculus through Fresh-Register Automata

Jevgenijus Čistiakovas, Master in Computer Science

University of Dublin, Trinity College, 2022

Supervisor: Dr. Vasileios Koutavas

Mobile concurrent systems permeate the world. Their often great importance means that errors in their designs can carry significant consequences. For guaranteeing correctness, formal verification methods are used. One particular approach is equivalence checking, where a design or implementation is checked against a specification.

The π-calculus is a language for describing mobile concurrent systems. The semantics of a π-calculus model is given by a labelled transition system (LTS) that describes how the model can transition states when interacting with the environment. Usually, such LTSs are infinite. However, the formalism of fresh-register automata (FRAs) can be used to represent many infinite-state models finitely.

This dissertation presents a solution to the problem of equivalence checking in π-calculus. It tackles the issue through an intermediate representation using the formalism of FRAs. The solution builds on prior work in generating LTSs of π-calculus models through the use of FRAs. To allow for equivalence checking in π-calculus an algorithm for n-bisimulation checking in the intermediate representation is proposed. The overall result corresponds to early bisimulation in π-calculus.

Furthermore, a command-line tool is created for practical equivalence checking of π-calculus models. The tool combines a modified version of the existing π-calculus-to-FRA translator with the implementation of the proposed algorithm. The tool supports both weak and strong early bisimulation checking. The tool is evaluated on the well-known examples from the literature and test cases from other similar tools. The results show that the approach is viable but is likely not practical for verifying large models.

Contents

List of Tables

List of Figures

Listings

List of Algorithms

Chapter 1

Introduction

1.1 Background and motivation

Mobile concurrent systems permeate the world. The best-known examples are found in the areas of distributed computing and operating systems. In there, systems often consist of multiple autonomous components that communicate with each other to create a coherent structure. And they are mobile as their communication topology can change dynamically. This mobility is usually implemented through reference passing: a node in a distributed system can discover a new connection by receiving an Internet Protocol (IP) address. Nevertheless, the systems with the given properties are not limited to the areas in classical computer science and also can be found, for example, in molecular biology.

There is an increasing reliance on such systems operating correctly, which means that design errors are highly undesirable. Defects can lead to significant monetary losses when, for example, a system is needed to operate a business. However, they can also lead to even more significant consequences in safety-critical systems such as nuclear power plants or aeroplane control systems.

The issue is that designing complex systems is hard. It is notoriously hard even for classical concurrent systems and even more so for mobile concurrent systems. Thus, various system verification techniques are used. A commonly used verification technique for limiting the number of errors is different testing methods such as unit testing or integration testing in the context of software systems. However, by definition, testing can only increase one's confidence that a design is correct but cannot guarantee it. For stronger guarantees, formal verification methods need to be used.

One particular approach to formal verification is equivalence checking. It is concerned with establishing whether two systems have the same behaviour. Traditionally, this is exactly the central issue in concurrency theory. There are several relations with different

properties that can be used. A commonly used equivalence is bisimulation and its weak variant - weak bisimulation. The latter is concerned with observable behaviour, which is of the main interest. In this framework, it is then possible to check for the correctness of a designed system. An implementation is correct if it has an equivalent behaviour to its specification. And assuming the specification is usually compact, its correctness is relatively easy to establish.

As mentioned, mobile concurrent systems can have many appearances. For the purpose of formal verification, it is desirable to have a modelling language in which a model for every possible system can be created. Specifically, in equivalence checking, it is desirable to use a single language to both specify the design and the specification of a system. There are two main aspects of modelling languages. On the one hand, great expressiveness is desirable as any conclusion derived from a model is only as good as the model itself. On the other hand, simplicity is needed to keep the theory tractable and verification feasible. In the context of sequential computation, λ-calculus is regarded as a canonical model and can be used to represent any computable function. However, it is unsatisfactory when dealing with concurrency.[35] For mobile concurrent systems, π-calculus is the closest to being a universal calculus for concurrent computations[26], and it is the topic of this dissertation.

The π-calculus is a simple language that achieves great expressiveness. It can, for example, fully model λ-calculus.[25] However, that also makes it intractable for automatic verification techniques. A common approach is to trade off some of the expressiveness for tractability. Nevertheless, regardless of that, the key challenge with π-calculus is that it works with infinite alphabets. Hence, even simple finite processes that receive an input from an environment lead to infinite execution graphs, which cannot be exhaustively checked as is needed for equivalence checking. This problem is naturally overcome with higher-level reasoning over names. A particular formalism that provides a solution to infinite names and is used in this dissertation is fresh-register automata (FRAs)[37].

From a practical standpoint, throughout the years π-calculus has been applied to several interesting use cases. It has found a great application in the area of verification of security protocols. An extension to π-calculus called applied π-calculus[1] was created, making it more convenient to reason about security protocols in the context of π-calculus. Verification with applied π-calculus is usually done in the tool ProVerif.[3] An example of a practical application would be the verification of a remote electronic voting protocol by Backes et al. using applied π-calculus and ProVerif.[2] Classical π-calculus has found application in the verification of a handover procedure in GSM Public Land Mobile Network.[31] The π-calculus has also found application in non-computer-science areas such as molecular biology[30] or business process engineering[33].

Overall, it has been more than thirty years since the creation of π-calculus. The problem of equivalence checking in π-calculus has been tackled using different approaches. However, no single solution has been accepted as entirely satisfactory. This dissertation considers an approach through a novel formalism of fresh-register automata.

1.2 Objectives

The main objective of this dissertation is to create an end-to-end equivalence checker for π-calculus models. More specifically, the objective is to investigate the possibility of using fresh-register automata for tackling the problem of equivalence checking in π-calculus.

The work in this dissertation builds on the dissertation of S. Leung with the title *Modelling Concurrent Systems: Generation of Labelled Transition Systems (LTSs) of Pi-Calculus Models through the Use of Fresh-Register Automata.*[18] Leung worked under the same supervisor - V. Koutavas - and produced a tool `pifra` for translating π-calculus models into fresh-register automata. This tool is central to the work in this dissertation. Leung's work is based on the paper *Fresh-Register Automata* by N. Tzevelekos[37], which is also the basis for this dissertation.

Overall, the objectives of the dissertation are as follows:

- Investigate the limitations of the previous attempt on a similar problem by B. Contovounesios. In his dissertation work [8], he also attempted to implement an equivalence checker for π-calculus models. However, due to various reasons, only minimal success was achieved.
- Design an algorithm for checking n-bisimulation for $\times\pi$-calculus models. The $\times\pi$-calculus is a language that is defined by Tzevelekos in his paper and which connects π-calculus and fresh-register automata. Meanwhile, n-bisimulation is a specific equivalence relation that connects equivalences in π-calculus and $\times\pi$-calculus.
- Implement a tool for equivalence checking of π-calculus models through bisimulation in $\times\pi$-calculus.
- Evaluate the practicality of using fresh-register automata for equivalence checking of π-calculus models.

To the author's best knowledge, there exists neither an algorithm for n-bisimulation nor a tool for equivalence checking (early bisimulation) via the formalisms of fresh-register automata and $\times\pi$-calculus.

1.3 Contributions

The contributions of this dissertation can be summarised as follows:

- A novel algorithm for checking n-bisimulation between $\times\pi$-calculus models. This new algorithm extends an existing on-the-fly algorithm for classical bisimulation between LTSs to handle n-bisimulation in $\times\pi$-calculus. To the best knowledge of the author, this is the first such algorithm.
- A tool `pisim22` for end-to-end equivalence checking of π-calculus models. This tool can check for both strong and weak early bisimulation. The tool tackles the bisimulation problem in π-calculus through the intermediated use of FRAs formalism, and the n-bisimulation algorithm mentioned previously.

  In particular, the tool created re-uses another tool `pifra` by Leung.[18] His tool allows for translation from π-calculus to $\times\pi$-calculus LTSs, and is thus used as a front-end for `pisim22`. However, Leung's tool had to be extended and corrected to make it suitable for use in `pisim22`. Otherwise, the implementation of the n-bisimulation algorithm is fully due to the author. The implementation of transformation necessary for weak bisimulation is due to the author and is based on the ideas from Cleaveland and Sokolsky in [7].

  To the best knowledge of the author, this is the first implementation of bisimulation checking in π-calculus through FRA and $\times\pi$-calculus.

1.4 Dissertation structure

This dissertation has been written in a style that does not assume any prior knowledge of concurrency theory in general and π-calculus in particular. No knowledge of formal verification and equivalence checking is assumed.

The dissertation is structured as follows:

- Chapter 1 - Introduction – gives an overview and motivation for the topic of this dissertation and lists the objectives and contributions of the work.
- Chapter 2 - Background – gives an in-depth overview of all the necessary background for understanding the problem and the work in this dissertation, including the basics of labelled transition systems and bisimulation, the π-calculus and observational equivalence, the fresh-register automata, the $\times\pi$-calculus and n-bisimulation. Additionally, an overview of related work is provided. In particular, a review of the previous work on the topic and its limitations is given.

- Chapter 3 - Design – describes the n-bisimulation algorithm, one of this dissertation's main contributions. The chapter provides the background on Celikkan's algorithm, on which the approach builds, and examines the correctness and complexity of the proposed new algorithm.
- Chapter 4 - Implementation – covers the details of the implemented `pisim22` tool, describing the functionality of the tool as well as covering the internal design and implementation details of the tool.
- Chapter 5 - Evaluation – describes how the tool was tested, additionally examining the performance of the tool, as well as comparing it to some other alternative solutions.
- Chapter 6 - Conclusion – summarises the achievements and discusses the limitations and future work.

Chapter 2

Background

This chapter introduces the background necessary for understanding the goals and contributions of this dissertation. In the first part, an overview of topics such as labelled transition systems, bisimulation, π-calculus, and fresh-register automata is provided. The second part of the chapter provides an overview of related work. The previous projects on which this dissertation builds are described. In particular, limitations of the previous attempt at the same problem are reported.

2.1 Labelled transition systems

The most basic and vital concept relevant to this dissertation is that of a labelled transition system.

Definition 2.1 Labelled transition system (LTS). [7, definition 2.1]

A labelled transition system (LTS) is a triple $\langle S, A, \rightarrow \rangle$, where S is a set of states, A is a set of actions, and $\rightarrow \subseteq S \times A \times S$ is the transition system. $\Diamond$

Intuitively, an LTS $\langle S, A, \rightarrow \rangle$ represents a general process of computation. The set S contains all the states in which the system can be, the set A represents all the actions the system might engage in, and the relation $\rightarrow$ defines what actions the system can take from a specific state and what state the system will end up in by taking one of the possible actions. As the definition of the LTS does not put any restrictions on the sets S, A or relation $\rightarrow$, it is a very general concept. Hence, it can be used to represent various automata. Moreover, a single LTS can represent multiple automata.

In the rest of this dissertation, the relation $(s, a, s') \in \rightarrow$ will often be written as $s \xrightarrow{a} s'$. Labelled transitions systems can be visualised as directed, edge-labelled graphs.

Example 2.2. Consider an example of an LTS in Figure 2.1. $\triangle$

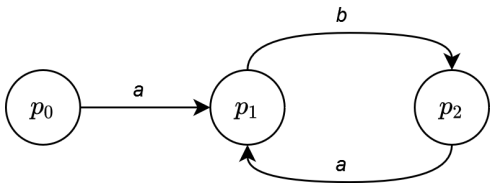

Figure 2.1: Example of a simple LTS with states $S = \{p_0, p_1, p_2\}$, actions $A = \{a, b\}$ and transitions given by arrows in the figure.

2.2 Strong bisimulation

The central topic of this dissertation is determining if two systems are equivalent, where a system can be represented as an LTS. There exist many equivalences with different properties. The equivalence relation used in this dissertation is *bisimulation*. It achieves a great balance of strength by, for example, not being as limiting as a graph isomorphism which is too concerned with the exact shape of an LTS and being stronger than trace equivalence, which is a similar concept to language equivalence in automata theory. Bisimulation also exhibits such desirable properties as a locality of checks and a lack of temporal order on the checks.[35][34]

In this dissertation, bisimulation (or strong bisimulation) will be defined through the notion of simulation (or strong simulation).

Definition 2.3 Strong simulation. Based on [25, p. 17, definition 3.3] and [7, definition 2.8] Let $\langle S, A, \rightarrow \rangle$ be an LTS. A binary relation $\mathcal{R}$ on the states of the LTS is a *strong simulation* (or simply a simulation) if when $(s_1, p_1) \in \mathcal{R}$ then the following holds for all $a \in A$:

- If $s_1 \xrightarrow{a} s_2$, then there is $p_2 \in S$ such that $p_1 \xrightarrow{a} p_2$ and $(s_2, p_2) \in \mathcal{R}$.

A state p is said to simulate s if there exists a strong simulation $\mathcal{R}$ such that $s\mathcal{R}p$. $\Diamond$

It is then possible to define bisimulation as follows:

Definition 2.4 Strong bisimulation. Based on [25, p. 18, definition 3.6]

Let $\langle S, A, \rightarrow \rangle$ be an LTS. A binary relation $\mathcal{R} \subseteq S \times S$ is a *strong bisimulation* (or simply a bisimulation) if both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are strong simulations. Two states $s, p \in S$ are strongly bisimilar or bisimulation equivalent, written as $s \sim p$, if there exists a strong bisimulation $\mathcal{R}$ such that $s\mathcal{R}p$. $\Diamond$

Note that bisimulation is a stronger condition than simulation in both directions. That is, it is possible that p simulates q under simulation R and that in the reverse direction, q simulates p under simulation S, but that p and q are not bisimilar because $S \neq R^{-1}$ for all possible S and R for a specific LTS.

To aid the understanding of the algorithm proposed in this dissertation, it is useful also to consider a more self-contained definition of bisimulation.

Definition 2.5 Bisimulation, bisimulation equivalence. [7, definition 2.2] Let $\langle S, A, \rightarrow \rangle$ be an LTS.

- A relation $\mathcal{R} \subseteq S \times S$ is a *bisimulation* if whenever $(s_1, p_1) \in \mathcal{R}$ then the following hold for all $a \in A$:
  1. If $s_1 \xrightarrow{a} s_2$, then there is p_2 such that $p_1 \xrightarrow{a} p_2$ and $(s_2, p_2) \in \mathcal{R}$.
  2. If $p_1 \xrightarrow{a} p_2$, then there is s_2 such that $s_1 \xrightarrow{a} s_2$ and $(s_2, p_2) \in \mathcal{R}$.
- Two states, $s_1, p_1 \in S$, are *bisimulation equivalent*, written as $s \sim p$, if there exists a strong bisimulation $\mathcal{R}$ such that $s\mathcal{R}p$

$\Diamond$

Example 2.6 Simulation and bisimulation. Consider the LTS in Figure 2.2. In this LTS, state p_0 is bisimilar to q_1 as there exists bisimulation relation R with $p_0 R q_1$ given by:

$$R = \{(p_0, q_1), (p_1, q_2), (p_2, q_1)\}$$

Note that this is a single LTS with five states, even though there are two distinct subgraphs. It is important as bisimulation is defined for states within a single LTS. However, in practice, it is often desirable to check two different LTSs for equivalence, and in this case, they need to be merged, implicitly or explicitly. $\triangle$

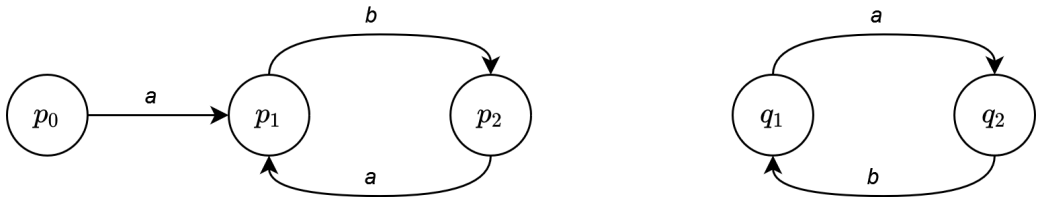

Figure 2.2: Example of bisimilarity. States p_0 and q_1 are bisimilar.

2.3 Bisimulation algorithms

As mentioned in the Introduction chapter, the ability to practically check for bisimulation equivalence is of great interest as it allows for the verification of systems.

On a high level, there are two traditional families of equivalence checking algorithms. *Global algorithms* require having a full LTS before running the algorithm. In contrast, *local algorithms*, also known as on-the-fly algorithms, construct an LTS incrementally while running the algorithm.

Global algorithms are more well established. It is partly because the bisimulation problem is equivalent to a well studied *coarsest state partition problem.*[14] The algorithms that solve this problem are known as partition refinement algorithms.[7] The best-known algorithms are due to Kanellakis and Smolka [17] and Paige and Tarjan [32]. Given an LTS $\langle S, A, \rightarrow \rangle$, where $|S| = n$ and $| \rightarrow | = m$, the Kanellakis-Smolka algorithm takes time in $O(n * m)$, while the Paige-Tarjan algorithm improves the time to be in $O(m * log(n))$.

The main drawback of the partition refinement algorithms is that they require all the information about an LTS before executing an algorithm. Computing all the states and transitions prior to checking can be very expensive, particularly from a memory standpoint. On the other hand, computing the information dynamically, like in local algorithms, inevitably incurs a time penalty. A slightly orthogonal to these approaches are *symbolic algorithms.* They can be global or local, with the key property of succinctly representing the LTS. An example of those is the local algorithm by Bouali and de Simone that uses ordered binary decision diagrams (OBDDs) as basic data structures.[4]

Local algorithms do not need to know a full LTS at the start of the execution. Instead, they can generate the transitions and necessary states as they execute. Local algorithms perform the best in scenarios when the full LTS is not known beforehand, when there is limited memory and when the states checked are likely not to be bisimilar. The idea behind most of local algorithms is reminiscent of the game-theoretic formulation of bisimulation. The algorithms start with a pair of states to be checked for bisimilarity and proceed in a depth-first search (DFS) fashion to explore and match the transitions until either a counter-example is found or a bisimulation relation is built. Various strategies such as stateful backtracking are used to allow for non-determinism. Examples of local algorithms include the algorithm by Fernandez and Mounie [12], the algorithm by Celikkan presented by Cleaveland and Sokolsky[7]. Additionally, Lin has lifted the on-the-fly bisimulation algorithm to handle value-passing processes, hence employing a local algorithm at a symbolic level.[19] In terms of complexity, the algorithm by Celikkan takes $O((n+m)^2)$ time.

In practice, all varieties of algorithms have been used for automatic bisimulation checking, and some were applied for equivalence checking of concurrent systems. Global algorithms have been used in tools such as Aldébaran [10] and the Concurrency Workbench [6]. In practice, local algorithms have been used in tools such as the Mobility Workbench [38], and Aldébaran [11].

2.4 The π-calculus

The main object of interest in this dissertation is π-calculus. It is a process algebra designed for describing mobile processes. That is, it is a formal language for modelling mobile concurrent systems with transitional semantics defined that describe how concurrent components of a system can interact with each other and the environment.[26]

The π-calculus was designed with the idea of it being a universal calculus for concurrent computations, similar to how λ-calculus is for functional computation.[26] As a result, it possesses great expressiveness. For example, in their books on π-calculus, both Milner [25] and Sangiorgi [36] demonstrate how π-calculus can be used to model data structures such as lists, functional computation through λ-calculus, and object-oriented programming.

There exist slightly different definitions of π-calculus. However, these variations usually do not affect the expressiveness of the calculus. In this dissertation, the grammar presented is based on the classical definition from Milner's original paper [26] with some cosmetic changes that were commonly used by Milner himself [25] and other authors. The grammar for π-calculus is then defined as presented below.

Definition 2.7 The π-calculus. [26][37] Let $\mathcal{N}$ be an infinite set of *channel names* with lower case letters such as $x, y, z \in \mathcal{N}$ used as metavariables over names. Also, assume the existence of a set of *process identifiers* with p ranging over that set. Each process identifier has a non-negative arity, and the arity of $p(x_1, ..., x_n)$ is n. The set Π of π-calculus processes is then defined by the following grammar (given in Backus-Naur form):

$P, Q ::=$		*process*
	0	*inaction*
|	$\overline{x}\langle y\rangle.P$	*output*
|	$x(y).P$	*input*
|	$\tau.P$	*unobservable action*
|	$\nu x.P$	*restriction*
|	$[x = y]P$	*match*
|	$P + Q$	*summation*
|	$P \mid Q$	*composition*
|	$p(x_1, ..., x_n)$	*process call*

◊

The interpretation of the above is as follows:

- P and Q are processes. The most basic process is a nullary process 0, which represents inaction. It is a final expression in any terminating process and thus is often omitted.
- The capability to do computation is expressed by *action prefixes*: $\overline{x}\langle y\rangle$, $x(y)$, and τ. The output action prefix $\overline{x}\langle y\rangle.P$ denotes the action of sending the name y on the channel x and then continuing as P. The input action prefix $x(y).P$ denotes the action of receiving any name on the channel x and then continuing as P with the received name substituted for y in P. Finally, the unobservable action prefix $\tau.P$ denotes a possibility of $\tau.P$ silently evolving into P. The unobservable actions can be thought to represent an internal activity of a process.
- The match expression $[x = y]P$ represents an ability to do conditional action. The action in P can only be taken if the names x and y are the same.
- The summation and composition expressions represent the ways of combining multiple processes together. Summation $P + Q$ represents a non-deterministic choice, while composition $P \mid Q$ represents concurrent execution.
- The restriction operator νx controls the scope of the name x. In expression $\nu x.P$, the scope of the name x is restricted to P, which means that components within P can use the channel x to interact with each other, but it is not allowed to use the name x outside of the boundaries of P.

  Directly related to this is the notion of binding. Input and restriction operators, $x(y).P$ and $\nu y.P$, bind the name y with scope P, and such occurrences are called binding occurrences. Meanwhile, an occurrence of y in a process is said to be *free* if it is not within the scope of a binding occurrence. For a given process Q, a name can be bound or free. The set of names appearing in Q that are free is denoted as $fn(Q)$. The set of bound names is denoted as $bn(Q)$.
- The process call expression denotes a parametrised process characterised by a process identifier and a list of names $(x_1, ..., x_n)$. Each process identifier has a corresponding defining equation of the form $p(y_1, ..., y_n) = P$, where individual y_i are pairwise distinct and $fn(P) \in \{y_1, ..., y_n\}$. And then, semantically, the process call represents a process P with each x_i substituted for y_i.

The semantics of the calculus is given via an LTS, where processes are states and actions arise from the action prefixes and are of the types $a ::= \overline{x}y \mid \overline{x}(y) \mid xy \mid \tau$. The first action is sending a name y over the channel x, the second action is sending a fresh name y over x, the third is receiving y over x, and the last one is an internal action. A transition from process P_1 to process P_2 by taking an action α is denoted as $P_1 \xrightarrow{\alpha} P_2$.

Actions also have free and bound occurrences of names. The only bound occurrence is in $\overline{x}(y)$ actions where y is bound. All the other names in $\overline{x}(y)$ and $\overline{x}y$, and xy are free, and there are no names in τ action. The notations follows the one for names in processes. Additionally, the set of names is defined as $n(\alpha) = bn(\alpha) \cup fn(\alpha)$.

The exact semantics of transitions is defined by a set of rules in table 2.1 (symmetric rules for SUM, PAR, COMM, CLOSE omitted).

$$\text{TAU}\ \frac{}{\tau.P \xrightarrow{\tau} P} \qquad \text{CLOSE\_L}\ \frac{P \xrightarrow{\overline{x}(y)} P' \qquad Q \xrightarrow{xy} Q'}{P \mid Q \xrightarrow{\tau} \nu y.(P' \mid Q')}\ y \notin fn(Q)$$

$$\text{OUT}\ \frac{}{\overline{x}y.P \xrightarrow{\overline{x}y} P} \qquad \text{MATCH}\ \frac{P \xrightarrow{\alpha} P'}{[a=a]P \xrightarrow{\alpha} P'}$$

$$\text{IN}\ \frac{}{x(y).P \xrightarrow{xw} P\{w/y\}} \qquad \text{REC}\ \frac{P\{\vec{x}/\vec{y}\} \xrightarrow{\alpha} P'}{p(\vec{x}) \xrightarrow{\alpha} P'}\ p(\vec{y}) = P$$

$$\text{OPEN}\ \frac{P \xrightarrow{\overline{x}y} P'}{\nu y.P \xrightarrow{\overline{x}(y)} P'}\ x \neq y \qquad \text{RES}\ \frac{P \xrightarrow{\alpha} P'}{\nu a.P \xrightarrow{\alpha} \nu a.P'}\ a \notin n(\alpha)$$

$$\text{SUM\_L}\ \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \text{COMM\_L}\ \frac{P \xrightarrow{\overline{x}y} P' \qquad Q \xrightarrow{xy} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$\text{PAR\_L}\ \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}\ bn(\alpha) \cap fn(Q) = \emptyset$$

Table 2.1: Action rules.

The following example attempts to illustrate the action rules in practice.

Example 2.8 Demonstrating π-calculus action rules. Consider the following process:

$$P = i(x).\overline{o}\langle x\rangle.0 + \nu c.\big(i(x).\overline{c}\langle x\rangle.0 \mid c(x).\overline{o}\langle x\rangle.0\big)$$

It is effectively a process identifier of arity 0. The set of free names of P is $fn(P) = \{i, o\}$. The other names are either bound by restriction such as c or bound by input such as x. As mentioned, it can be rewritten with nullary processes removed for clarity:

$$P = i(x).\overline{o}\langle x\rangle + \nu c.\big(i(x).\overline{c}\langle x\rangle \mid c(x).\overline{o}\langle x\rangle\big)$$

There are three top level actions in P: $i(x)$, $i(x)$, and $c(x)$. The only possibility to

transition from P is to accept an input on the channel i. As the channel name is free, and there is no complementary output action on channel i within the P, the message can only come from the outside context. On the other hand, P cannot accept an input on channel c. The reason is that since c has a restricted scope, it can only interact with a complementary $\overline{c}$, and there is no top level output action for that in P. Thus, there are two possibilities of transition on input action $i(x)$, and with the summation operator the choice will be non-deterministic. Assume that the name received is y and hence the action is $i(y)$. The possibilities are then $P \xrightarrow{i(y)} P_1$ or $P \xrightarrow{i(y)} P_2$, where

$$P_1 = \overline{o}\langle y\rangle$$
$$P_2 = \nu c.\big(\overline{c}\langle y\rangle \mid c(x).\overline{o}\langle x\rangle\big)$$

From P_1, there is only one possible transition: $P_1 \xrightarrow{\overline{o}\langle y\rangle} P_3$. For P_2, the only possible transition is the reaction between c and $\overline{c}$. This is possible since c and $\overline{c}$ are in parallelly running processes. Individually these subprocesses transition as follows: $\overline{c}\langle y\rangle \xrightarrow{\overline{c}\langle y\rangle} 0$ and $c(x).\overline{o}\langle x\rangle \xrightarrow{c(y)} \overline{o}\langle y\rangle$. But at the level of P_2, this is an internal action represented by transition $P_2 \xrightarrow{\tau} P_1$, where

$$P_3 = 0$$

Lastly, P_4 can possibly perform an output action to the context. The possible transition is then $P_4 \xrightarrow{\overline{o}\langle y\rangle} P_3$.

Overall, the two transition paths are $P \xrightarrow{i(y)} P_1 \xrightarrow{\overline{o}\langle y\rangle} P_3$ and $P \xrightarrow{i(y)} P_2 \xrightarrow{\tau} P_1 \xrightarrow{\overline{o}\langle y\rangle} P_3$. The only difference between the paths is a presence of an internal transitions τ in the latter. See a schematic of an LTS in Figure 2.3. This LTS is finite as it only shows one possible input of $i(y)$. However, since π-calculus operates on an infinite alphabet of names, there could be an infinite number of possible inputs from the context: $i(b) \mid b \in \mathcal{N}$. As the result, an actual LTS for the considered process would be infinite.

△

Note that there is a choice of when the instantiation of a variable occurs. For example, consider a process $a(x).P$. With *early instantiation* the variable x will be instantiated immediately at the time when the input transition occurs. This means that the input action carries the name received and not the variable name. In contrast, with *late instantiation*, the instantiation will be delayed until after the transition.[26] This choice and other details affect the exact definition of transition rules and bisimulation. In this work, early semantics is used, and thus early bisimulation is assumed where relevant.

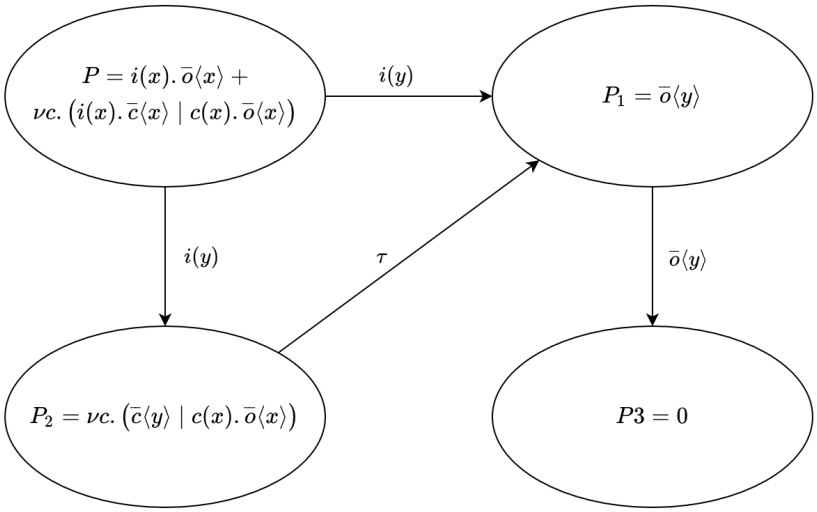

Figure 2.3: A finite part of the LTS for π-calculus process corresponding to one possible input $i(y)$.

2.4.1 Bisimulation in π-calculus

The definitions of simulation and bisimulation for π-calculus are extensions of the ones for a generic LTS. This is possible as the calculus semantics is given via an LTS. For a process P, consider an LTS $\langle S, A, \rightarrow\rangle$, where S is a set of all processes that P can evolve into as defined by the action rules, A is a set of actions of type $\alpha ::= \overline{x}y \mid \overline{x}(y) \mid xy \mid \tau$ with $x, y \in \mathcal{N}$, and $\rightarrow$ is given by the generating action rules. Given that LTS, simulation and bisimulation can be defined as follows.

Definition 2.9 Strong simulation in π-calculus. [37] A relation $\mathcal{R} \subseteq \Pi \times \Pi$ is called a *strong simulation* (or just simulation) if, for all $(P_1, P_2) \in \mathcal{R}$ and all action labels α with $bn(\alpha) \cap fn(P_1, P_2) = \emptyset$, if $P_1 \xrightarrow{\alpha} P_1'$ then $P_2 \xrightarrow{\alpha} P_2'$ for some $(P_1', P_2') \in \mathcal{R}$. ◊

Definition 2.10 Strong bisimulation in π-calculus. [37] A relation $\mathcal{R} \subseteq \Pi \times \Pi$ is called a *strong bisimulation* (or just bisimulation) if both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are strong simulations. Processes $P, Q \in \Pi$ are then said to be π-bisimilar, written as $P \overset{\pi}{\sim} Q$, if there is a bisimilation $\mathcal{R}$ containing (P, Q). ◊

The problem of checking bisimulation in π-calculus is central to this dissertation. The challenge arises from the fact that usage of regular LTS and classical bisimulation algorithms is very limiting due to the set of channel names being infinite. In this case, even very simple processes that interact with the context through free channels lead to an LTS with infinite branching, one per each name in the infinite set of names. For an example of such a process, see Example 2.8. For this reason, more successful approaches

treat the names symbolically. That is, representing the names more succinctly with a symbolic variable and some constraints instead of an actual name.

2.5 Weak bisimulation

It is often helpful to treat the systems, such as those specified in π-calculus, as black boxes. That is, to consider all the internal actions as being transparent, and pay attention only to interactions with the environment that are visible to an outside observer. The corresponding equivalence is called an observational equivalence, and the commonly used relation is weak bisimulation.

Central to weak bisimulation is the distinction between internal (silent) actions denoted as τ and observable actions (non-τ) denoted as a. Weak bisimulation is concerned with only matching observable actions. To be able to define weak bisimulation, consider the following notation, which is an extension to a transition function $\rightarrow$.

Definition 2.11 Weak transition. [35, p. 110, definition 4.1.1] Let $\langle S, A, \rightarrow \rangle$ be an LTS, where τ is an internal action.

- Relation $\Rightarrow \in S \times S$ is the reflexive and transitive closure of $\xrightarrow{\tau}$. That is, $P \Rightarrow P'$ holds if there is $n \geq 0$ and processes $P_1, ..., P_n$ with $P_n = P'$ such that $P \xrightarrow{\tau} P_1 \xrightarrow{\tau} ... \xrightarrow{\tau} P_n$.
- For all $a \in A$, relation $\stackrel{a}{\Rightarrow} \in S \times S$ is the composition of the relations $\Rightarrow$, $\xrightarrow{a}$, and $\Rightarrow$. That is, $P \stackrel{a}{\Rightarrow} P'$ holds if there are P_1, P_2 such that $P \Rightarrow P_1 \xrightarrow{a} P_2 \Rightarrow P'$.

$\Diamond$

Example 2.12 Weak transitions in π-calculus. Consider the π-calculus process from Example 2.8. To an external observer both possible interactions appear the same. Using weak transitions, $P \xrightarrow{i(y)} P_1 \xrightarrow{\overline{o}\langle y \rangle} P_3$ can be written as $P \stackrel{i(y)}{\Longrightarrow} P_1 \stackrel{\overline{o}\langle y \rangle}{\Longrightarrow} P_3$, while $P \xrightarrow{i(y)} P_2 \xrightarrow{\tau} P_1 \xrightarrow{\overline{o}\langle y \rangle} P_3$ can be written as either $P \stackrel{i(y)}{\Longrightarrow} P_1 \stackrel{\overline{o}\langle y \rangle}{\Longrightarrow} P_3$ or $P \stackrel{i(y)}{\Longrightarrow} P_2 \stackrel{\overline{o}\langle y \rangle}{\Longrightarrow} P_3$. Notice how the labels of the weak transitions are the same in all the cases. $\triangle$

While there exist multiple variations of weak bisimulation, all the main authors such as Milner[25, p. 54, defintion 6.5], Cleaveland and Sokolsky[7, definition 3.3], Sangiorgi [35, p.110, definition 4.2.1] agree on one common definition that is also called weak bisimulation.

Definition 2.13 Weak simulation. [25, p. 53, definition 6.2] Let $\langle S, A, \rightarrow \rangle$ be an LTS. A relation $\mathcal{R} \in S \times S$ is a *weak bisimulation* if whenever $s_1 \mathcal{R} p_1$, then

- if $s_1 \Rightarrow s_2$ then there exists $p_2 \in S$ such that $p_1 \Rightarrow p_2$ and $s_2 \mathcal{R} p_2$.

- if $s_1 \stackrel{a}{\Rightarrow} s_2$ then there exists $p_2 \in S$ such that $p_1 \stackrel{a}{\Rightarrow} p_2$ and $s_2 \mathcal{R} p_2$.

$\Diamond$

Furthermore, weak bisimulation can be re-defined in a way that is more suitable for computation.

Definition 2.14 Weak simulation simplified. [25, p. 53, proposition 6.3] Let $\langle S, A, \rightarrow \rangle$ be an LTS, where τ is an internal action. A relation $\mathcal{R} \in S \times S$ is a weak bisimulation if and only if, whenever $s_1 \mathcal{R} p_1$, then

- if $s_1 \stackrel{\tau}{\rightarrow} s_2$ then there exists $p_2 \in S$ such that $p_1 \Rightarrow p_2$ and $s_2 \mathcal{R} p_2$.
- if $s_1 \stackrel{a}{\rightarrow} s_2$ then there exists $p_2 \in S$ such that $p_1 \stackrel{a}{\Rightarrow} p_2$ and $s_2 \mathcal{R} p_2$.

$\Diamond$

Weak bisimulation definition then follows naturally as:

Definition 2.15 Weak bisimulation. [25, p. 54, defintion 6.5] Let $\langle S, A, \rightarrow \rangle$ be an LTS. A relation $\mathcal{R} \in S \times S$ is said to be a *weak bisimulation* if both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are weak simulations. Two states $s, p \in S$ are said to be weakly bisimilar, weakly equivalent, or observation equivalent, written as $s \approx p$, if there exists a weak bisimulation R such that sRp. $\Diamond$

The definition above naturally extends to all the more specific versions of strong bisimulation, such as bisimulation for π-calculus. Note, however, that in π-calculus weak bisimulation is not preserved by the composition operator. E.g. it holds that $\tau.P \approx P$, but $\tau.P + Q \not\approx P + Q$.

2.5.1 Algorithms for weak bisimulation

To automatically check for weak bisimulation, the classical strong bisimulation algorithms can be used on a weakly-transformed LTS, which is defined as follows:

Definition 2.16 Weakly-transformed LTS. [7] Let $\langle S, A, \rightarrow \rangle$ be an LTS, where τ is an internal action. A weakly-transformed LTS is a triple $\langle S', A', \rightarrow' \rangle$, where $S' = S$, $A' = A - \{\tau\}$, and $\rightarrow'$ defined as follows:

- $s \stackrel{\tau}{\rightarrow}' s'$ when $s \Rightarrow s'$.
- $s \stackrel{a}{\rightarrow}' s'$ when $s \stackrel{a}{\Rightarrow} s'$.

$\Diamond$

Weakly-transformed LTS can be computed in two steps. First, a transitive closure on the τ-transitions of the original LTS is computed to obtain $\Rightarrow$. Second, $\stackrel{a}{\Rightarrow}$ is obtained by composing $\stackrel{a}{\rightarrow}$ with $\Rightarrow$ for all $a \neq \tau$ as per the definition of $\stackrel{a}{\Rightarrow}$.

2.6 Fresh-Register Automata

Previously, in Section 2.4.1, it was mentioned how using naive bisimulation checking approaches with π-calculus is unsatisfactory as even simple π-calculus processes result in infinite LTSs. Thus, a higher level symbolic reasoning is necessary.

There exists a successful model of computation that works with infinite alphabets - a finite-memory automaton (FMA) by Kaminski and Francez, which is equivalent to and synonymous to register automaton (RA).[16] The key idea behind FMA that allows it to work on infinite alphabets is an addition of a set of memory registers to a simple finite-state automaton. Each register can store a name. The automaton then transitions on labels which are defined to be the indices of the registers. Thus, previously there was an infinite number of labels corresponding to the infinite name set, while with FMA, the label set is finite and restricted by the size of the register set.

Tzevelekos further extended FMA by introducing an idea of a *globally fresh name*.[37] FMA is only capable of knowing whether an input name is stored in any of the registers or not - hence, determining if a name is *locally fresh*. On the other hand, Tzevelekos' fresh-register automaton (FRA) adds history tracking to FMA. Hence, FRA remembers the set of all the names seen so far and thus is capable of additionally recognising *global freshness* of a name. That is, whether a name has ever been seen by the automaton before. Both local and global freshness are important in order to be able to represent π-calculus. This additional feature allows FRAs to be able to represent π-calculus, which through the restriction expression supports the concept of globally fresh names.

Definition 2.17 Fresh-register automaton. [37, definition 1]

Let $\mathbb{A}$ be an infinite set of *names*. Let $\mathbb{L}_n$ be a set of labels generated by the set $[n] = \{1, ..., n\}$ for $n \in \omega$ (ordinal number), which is defined as

$$\mathbb{L}_n = \{i, i^\bullet, i^\circledast \mid i \in [n]\}$$

The register set is defined through a register assignment, which for size n is given as:

$$Reg_n = \{\sigma : [n] \to \mathbb{A} \cup \{\sharp\} \mid \forall i \neq j . \sigma(i) = \sigma(j) \implies \sigma(i) = \sharp\}$$

where $\sharp$ is a special symbol denoting an empty register.

A *fresh-register automaton (FRA)* of n registers is then a quintuple $\mathcal{A} = \langle Q, q_o, \sigma_0, \delta, F \rangle$ where:

- Q is a finite set of states,
- q_o is the initial state,
- $\sigma_o \in Reg_n$ is the initial register assignment,

- $\delta \subseteq Q \times \mathbb{L}_n \times Q$ is the transition relation,
- $F \subseteq Q$ is the set of final states.

$\Diamond$

Informally, an FRA $\mathcal{A} = \langle Q, q_o, \sigma_0, \delta, F \rangle$ is an automaton that starts in a state q_0 with an initial register assignment σ_0. Final states F determine which language is accepted by the automaton. During operation, on each input b, the automaton can transition into a new state by following the transition relation with the following interpretation[37]:

- **Known transition:** If $(q_1, i, q_2) \in \delta$, also written as $q_1 \xrightarrow{i} q_2$, and b is present in the register at index i, i.e. $\sigma(i) = b$, then $\mathcal{A}$ accepts b and moves to q_2.
- **Locally fresh transition:** If $q_1 \xrightarrow{i^\bullet} q_2 \in \delta$, and b is not present in σ, then $\mathcal{A}$ accepts b, stores b in a register by setting $\sigma(i) = b$, and moves to q_2.
- **Globally fresh transition:** If $q_1 \xrightarrow{i^\circledast} q_2 \in \delta$, and b was not part of the initial assignment σ_0 and has not appeared in the current run, then $\mathcal{A}$ accepts b, stores b in a register by setting $\sigma(i) = b$, and moves to q_2.

Naturally, an FRA can be represented as an LTS.

2.6.1 The $\times\pi$-calculus

The goal, for the purpose of this dissertation, is to have π-calculus represented by FRAs. To achieve that, Tzevelekos came up with an extended version of π-calculus that is called $\times\pi$-calculus with the key property that $\times\pi$-calculus is directly representable by FRAs. The motivation for extending π-calculus to $\times\pi$-calculus is that actions in π-calculus contain multiple names, whereas an FRA can only recognise one name at a time.

Definition 2.18 The $\times\pi$-calculus. [37, definition 29] The $\times\pi$-calculus syntax is given by the sets Π, Π_{out}, and Π_{inp} with elements defined as:

$$P, Q ::= \mathbf{0} \mid \overline{a}b.P \mid a(b).P \mid [a = b]P \mid \nu a.P \mid P + Q \mid (P|Q) \mid p(a_0, ..., a_n)$$
$$P_{out} ::= b.P \mid \nu a.P_{out} \mid (P|P_{out}) \mid (P_{out}|P)$$
$$P_{inp} ::= (b).P \mid \nu a.P_{inp} \mid (P|P_{inp}) \mid (P_{inp}|P)$$

$\Diamond$

The syntax of the main construction Π in $\times\pi$-calculus is essentially the same as the set of processes in π-calculus. The key difference is the addition of intermediate constructions Π_{out} and Π_{inp} that allow splitting the multi-named actions from π-calculus into single name transitions. Also, note that the definition of $\times\pi$-calculus here is missing the

explicit τ construct, but this is just how Tzevelekos defined it, and it does not reduce the expressiveness of the calculus. The semantics of the $\times\pi$-calculus still contain the τ action.

Definition 2.19 Semantics of $\times\pi$-calculus. [37, definition 30] The semantics of the $\times\pi$-calculus is given via an LTS with labels:

$$\alpha ::= i \mid i^{\bullet} \mid i^{\circledast} \mid \tau \mid \bar{i}j \mid \bar{i}j^{\circledast} \mid ij \mid ij^{\bullet}$$

where $i, j \in \omega$. The meaning of the labels $i^{\bullet}, i^{\circledast}$ is the same as in FRA.

The set of states is given by the set of processes-in-context $O(\hat{K})$ and it is a set of orbits for a nominal set $\hat{K}$. The set $\hat{K}$ in turn consists of all the possible pairs $(\sigma, \hat{P})$, where σ is a register assignment and $\hat{P} \in \Pi \cup \Pi_{out} \cup \Pi_{inp}$. Each such $O((\sigma, \hat{P}))$ is written as $\sigma \vdash \hat{P}$.

The transition relation is defined by a set of rules which can be seen in Table 2.2 (symmetric $PAR1$, $PAR2$, $COMM$, $CLOSE$ are omitted). The original rules were defined by Tzevelekos[37], but presented here is a modified set of rules with removed ambiguity due to Leung[18]. These rules naturally resemble those for π-calculus in Table 2.1. $\Diamond$

The set of transitions rules in Table 2.2 only lists double-label transitions. The conditions of the rules use the following notions:

- Image of a register assignment $\sigma \in Reg_n$ defined as $img(\sigma) = \{i \in [n] \mid \sigma(i) \in \mathbb{A}\}$.
- Label index $ind(\alpha) \subset \mathcal{P}(\mathbb{N})$ defined for labels as being $ind(\tau) = \emptyset$ and being $ind(\bar{i}j/\bar{i}j^{\circledast}/ij/ij^{\bullet}) = \{i, j\}$.
- Label symbol $sym(\alpha)$ defined as:

$sym(\tau)$	$sym(\bar{i}j)$	$sym(\bar{i}j^{\circledast})$	$sym(ij)$	$sym(ij^{\bullet})$
$\{\tau\}$	$\{\bar{i}, j\}$	$\{\bar{i}, j^{\circledast}\}$	$\{i, j\}$	$\{i, j^{\bullet}\}$

The key feature of the $\times\pi$-calculus is that the transition relation is finitely branching, which is not the case with regular π-calculus. The reasons are that, firstly, transitions are defined on labels that do not specify the channel name explicitly but instead specify the index of the register in which the name is stored or is to be stored. Secondly, the set of states consists of so-called orbits of the corresponding nominal set $\hat{K}$. Informally, this means that the set of states only contains a reduced set of equivalence classes, where elements in one equivalence class are the same up to a permutation of names. That is, for a pair $(\sigma, \hat{P})$ the exact names in σ or $\hat{P}$ are not important; instead, only the indices

$$\text{INP1}\ \frac{\sigma \vdash a(b).P \xrightarrow{i} \sigma \vdash (b).P \xrightarrow{j} \sigma \vdash P\{c/b\}}{\sigma \vdash a(b).P \xrightarrow{ij} \sigma \vdash P\{c/b\}}\ {\scriptstyle \sigma(i)=a \atop \sigma(j)=c}$$

$$\text{INP2}\ \frac{\sigma \vdash a(b).P \xrightarrow{i} \sigma \vdash (b).P \xrightarrow{j^\bullet} \sigma[j \mapsto b] \vdash P}{\sigma \vdash a(b).P \xrightarrow{ij^\bullet} \sigma[j \mapsto b] \vdash P}\ {\scriptstyle \sigma(i)=a \atop j=min\{j|\sigma(j)\notin fn(P)\} \atop b\notin img(\sigma)}$$

$$\text{OUT}\ \frac{\sigma \vdash \overline{a}b.P \xrightarrow{i} \sigma \vdash b.P \xrightarrow{j} \sigma \vdash P}{\sigma \vdash \overline{a}b.P \xrightarrow{\bar{i}j} \sigma \vdash P}\ {\scriptstyle \sigma(i)=a \atop \sigma(j)=b}$$

$$\text{RES}\ \frac{(\sigma + a) \vdash P \xrightarrow{\alpha} (\sigma' + a) \vdash P'}{\sigma \vdash \nu a.P \xrightarrow{\alpha} \sigma' \vdash \nu a.P'}\ {\scriptstyle (|\sigma|+1)\notin ind(\alpha)}$$

$$\text{OPEN}\ \frac{(\sigma + a) \vdash P \xrightarrow{\bar{i}j} (\sigma + a) \vdash P'}{\sigma \vdash \nu a.P \xrightarrow{\bar{i}k^\circledast} \sigma[k \mapsto a] \vdash \nu a.P'}\ {\scriptstyle i\neq j \atop j=(|\sigma|+1) \atop k=min\{i|\sigma(i)\notin fn(P')\} \atop a\notin img(\sigma)}$$

$$\text{MATCH1}\ \frac{\sigma \vdash P \xrightarrow{\alpha} \sigma' \vdash P'}{\sigma \vdash [a=a]P \xrightarrow{\alpha} \sigma' \vdash P'}$$

$$\text{REC}\ \frac{\sigma \vdash P\{\vec{a}/\vec{b}\} \xrightarrow{\alpha} \sigma' \vdash P'}{\sigma \vdash p(\vec{a}) \xrightarrow{\alpha} \sigma' \vdash P'}\ {\scriptstyle p(\vec{b})=P} \qquad \text{SUM}\ \frac{\sigma \vdash P \xrightarrow{\alpha} \sigma' \vdash P'}{\sigma \vdash P + Q \xrightarrow{\alpha} \sigma' \vdash P'}$$

$$\text{PAR1}\ \frac{\sigma \vdash P \xrightarrow{\alpha} \sigma \vdash P'}{\sigma \vdash P \mid Q \xrightarrow{\alpha} \sigma \vdash P' \mid Q}\ {\scriptstyle j^\bullet, j^\circledast \notin sym(\alpha)}$$

$$\text{PAR2}\ \frac{\sigma \vdash P \xrightarrow{ij^\bullet/\bar{i}j^\circledast} \sigma[j \mapsto b] \vdash P'}{\sigma \vdash P \mid Q \xrightarrow{ik^\bullet/\bar{i}k^\circledast} \sigma[k \mapsto b] \vdash P' \mid Q}\ {\scriptstyle k=min\{j|\sigma(j)\notin fn(P',Q)\}}$$

$$\text{COMM}\ \frac{\sigma \vdash P \xrightarrow{\bar{i}j} \sigma \vdash P' \qquad \sigma \vdash Q \xrightarrow{ij} \sigma \vdash Q'}{\sigma \vdash P \mid Q \xrightarrow{\tau} \sigma \vdash P' \mid Q'}$$

$$\text{CLOSE}\ \frac{(\sharp + \sigma) \vdash P \xrightarrow{\bar{i}1^\circledast} (b + \sigma) \vdash P' \qquad (\sharp + \sigma) \vdash Q \xrightarrow{i1^\bullet} (b + \sigma) \vdash Q'}{\sigma \vdash P \mid Q \xrightarrow{\tau} \sigma \vdash \nu b.(P' \mid Q')}$$

Table 2.2: Transitions relation for the $\times\pi$-calculus [18].

at which they are stored in σ matter. Note that this is a purely symbolic interpretation of the operation of the calculus.

Example 2.20. [37] The following two processes-in-context are the same up to a permutation of names:

$$\{(1,a),(2,c)\} \vdash a(b).\overline{b}\langle c\rangle.0 = \{(1,a'),(2,c')\} \vdash a'(b).\overline{b}\langle c'\rangle.0$$

with the following mappings $a \to a'$ and $c \to c'$. △

An LTS for $\times\pi$-calculus can be viewed in two ways. When restricted to single-label transitions, i.e. $\{i, i^{\bullet}, i^{\circledast}, \tau\}$, the resulting LTS contains states with processes in $\hat{P} \in \Pi \cup \Pi_{out} \cup \Pi_{inp}$. In this case, these is a straightforward direct mapping from the $\times\pi$-calculus to FRA. However, in this work π-calculus is of the main interest. Thus, the other way to build an LTS is to combine all the single-label transitions into double-label transitions, i.e. $\{\tau, \overline{i}j, \overline{i}j^{\circledast}, ij, ij^{\bullet}\}$ and restrict the processes to be in $P \in \Pi$. In this case, there is now an intuitive way of relating transitions of $\times\pi$-calculus and π-calculus as given by Tzevelekos in [37, lemma 32] and presented below:

Lemma 2.21. [37, lemma 32] Let σ, σ' be registers, and $\alpha, \hat{\alpha}$ be labels of π and $\times\pi$ respectively. For all processes $P, P' \in \Pi$ with $fn(P) \subseteq img(\sigma)$:

- if $\sigma \vdash P \xrightarrow{\hat{\alpha}} \sigma' \vdash P'$, then $P \xrightarrow{\alpha} P'$,
- if $P \xrightarrow{\alpha} P'$, then $\sigma \vdash P \xrightarrow{\hat{\alpha}} \sigma' \vdash P'$;

where either

- $\hat{\alpha} = \alpha = \tau$ and $\sigma = \sigma'$;
- or $\hat{\alpha} = \overline{i}j/ij$, $\alpha = \overline{a}b/ab$, $\sigma(i) = a, \sigma(j) = b$ and $\sigma' = \sigma$;
- or $\hat{\alpha} = \overline{i}j^{\circledast}/ij^{\bullet}$, $\alpha = \overline{a}(b)/ab$, $\sigma(i) = a, \sigma' = \sigma[j \mapsto b]$ and $j = min\{j \mid \sigma(j) \notin fn(P')\}$.

Remark. The above, together with the interpretation of FRAs, allows seeing how π-calculus is related to $\times\pi$-calculus and FRAs. Actions from $\times\pi$-calculus are represented as two-step (and τ) transitions in FRAs. For example, receiving a name over a channel is represented by FRA making a known transition on the channel name and then performing an either known, locally fresh or globally fresh transition based on the contents of the registers and history of previously seen names.

Example 2.22. Consider the following recursive π-calculus process: $P = a(x).(\nu y.\overline{x}\langle y\rangle.P)$, which is due to V.Koutavas.[18] The process first receives a name into variable x over channel a. It then generates a fresh name y and sends it over a previously received channel in

variable x. It calls itself recursively to execute these operations infinitely. This is clearly an infinite process that both can receive an infinite set of names on channel a as well as it will output an infinite number of globally fresh names. This process can be represented by a finite LTS when expressed in $\times\pi$-calculus. See Figure 2.4 for visualisation.

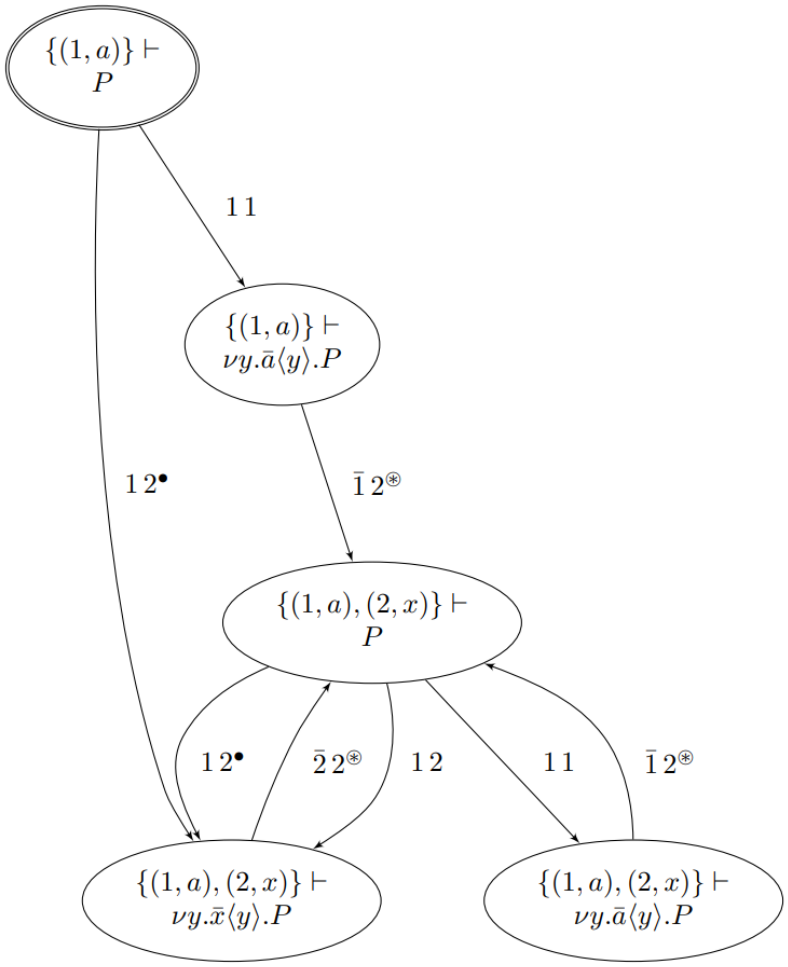

Figure 2.4: Example of a finite LTS when using $\times\pi$-calculus and FRAs.

Interpretation of the LTS is as follows. In the figure, the initial top-level process P is represented at the top with a double-lined border. The system has a register set of size two. Initially, only the first register contains the name a. This is the free name a for the input channel. From the starting state, the system can only accept input on this channel a. If the name received is the same as a, which is stored in the register, then the transition taken is 11. Otherwise, the name is said to be locally fresh, and the transition is $1\,2^{\bullet}$ with the new name x being put into the second register. The next action is to output a fresh name, this is represented by transitions $\bar{1}\,2^{\circledast}$ and $\bar{2}\,2^{\circledast}$. The rest of the transitions follow the same logic. Notice that the representation is finite and relatively compact. $\triangle$

The n-bisimulation

The last missing step is using the latter (double-label) LTS construction for bisimulation checking in $\times\pi$-calculus. For this, Tzevelekos defined a special notion of symbolic bisimulation that takes into account the multi-step transitions and makes a distinction between inputs and outputs. This relation is defined on states $O(K)$, where K consists of pairs (σ, P) where $P \in \Pi$; that is, only processes using the main constructions.

To define this new notion of bisimulation, the following notations will be used:

- Let $([n] \rightleftharpoons [n])$ be a set of typed spans on (n, n). Informally, this is a partial permutation which is a bijection between two (possibly different) subsets of $[n]$. A variable ρ will be used to represent an element of this set.
- Let $dom(\rho) \subseteq [n]$ be a domain of ρ defined as $dom(\rho) = \{j \in [n] \mid \exists j.(j, i) \in \rho\}$.
- Let $img(\rho) \subseteq [n]$ be an image of ρ defined as $img(\rho) = \{i \in [n] \mid \exists j.(j, i) \in \rho\}$.
- Let $\rho[i \leftrightarrow j]$, where $i, j \in [n]$, be a set $(\rho - \{(i', j') \mid i = i' \vee j = j'\}) \cup \{(i, j)\}$. That is, an update to a partial permutation adding a new mapping between i and j.
- Similarly, for register assignments $\sigma \in Reg_n$, let the domain $dom(\sigma)$ be the set of indices of all non-empty registers. Formally, $dom(\sigma) = \{i \in [n] \mid \sigma(i) \in \mathbb{A}\}$

Definition 2.23 The n-simulation. [37, definition 33] An *n-simulation* is a relation $R \subseteq O(K) \times ([n] \rightleftharpoons [n]) \times O(K)$. The conditions of the relation are that if $(\sigma_1 \vdash P_1, \rho, \sigma_2 \vdash P_2) \in R$ then $\sigma_1, \sigma_2 \in Reg_n$, and $\sigma_1 \vdash P_1 \xrightarrow{\alpha} \sigma_1' \vdash P_1'$ implies that there exists $\sigma_2 \vdash P_2 \xrightarrow{\alpha'} \sigma_2' \vdash P_2'$ and $(\sigma_1' \vdash P_1', \rho', \sigma_2' \vdash P_2') \in R$ such that one of the following is the case, with $i \in dom(\rho)$ (denoted as $NSYM$ rules):

- **TAU**: $\alpha = \alpha' = \tau$ and $\rho' = \rho$;
- **INP1**: $\alpha = ij, j \in dom(\rho)$, $\alpha' = \rho(i)\rho(j)$ and $\rho' = \rho$;
- **INP2**: $\alpha = ij, j \notin dom(\rho)$, $\alpha' = \rho(i)k^\bullet$ and $\rho' = \rho[j \leftrightarrow k]$, where $k \in [n]$;
- **FINP**: It is a conjunction of two conditions:
  - **FINP.1** $\alpha = ij^\bullet$, $\alpha' = \rho(i)k^\bullet$ and $\rho' = \rho[j \leftrightarrow k]$ and
  - **FINP.2** for all $k' \in dom(\sigma_2) - img(\rho)$ there exists $\sigma_2 \vdash P_2 \xrightarrow{\rho(i)k'} \sigma_2 \vdash P_2'$ and $(\sigma_1' \vdash P_1', \rho[j \leftrightarrow k'], \sigma_2 \vdash P_2') \in R$;
- **OUT**: $\alpha = \bar{i}j, j \in dom(\rho)$, $\alpha' = \overline{\rho(i)}\rho(j)$ and $\rho' = \rho$;
- **FOUT**: $\alpha = \bar{i}j^{\circledast}$, $\alpha' = \overline{\rho(i)}k^{\circledast}$ and $\rho' = \rho[j \leftrightarrow k]$.

$\Diamond$

Definition 2.24 The n-bisimulation. [37, definition 33] A relation R is called an *n-bisimulation* if both R and R^{-1} are n-simulations. Two processes P_1 and P_2 are

said to be n-bisimilar, written as $P_1 \overset{n}{\sim} P_2$, if there is an n-bisimulation R containing $(\sigma_{01} \vdash P_1, \sigma_{01} \leftrightarrow \sigma_{02}, \sigma_{02} \vdash P_2)$, for some $\sigma_{01}, \sigma_{02} \subseteq [n]$ where each σ_{01} and σ_{02} contain exactly all the free names in the corresponding processes. $\Diamond$

Theorem 2.25. *[37, proposition 34] For all processes P, Q whose all descendants have less than n free names it holds that $P \overset{\pi}{\sim} Q \iff P \overset{n}{\sim} Q$.*

The last theorem links together bisimulation in π-calculus and bisimulation in FRAs generated from $\times\pi$-calculus. Thus, a tool capable of checking for n-bisimulation would allow for checking π-calculus bisimulation, and it would decide a larger set of bisimulations compared to naive non-symbolic approaches.

Example 2.26 The n-bisimulation. Consider two processes: $P = c(x).b(x).0$ and $Q = (\nu x.\overline{x}\langle x\rangle.\overline{a}\langle x\rangle.0) + c(x).b(x).0$. Both are n-bisimilar for $n = 3$. The LTSs as generated from $\times\pi$-calculus are visualised in Figure 2.5.

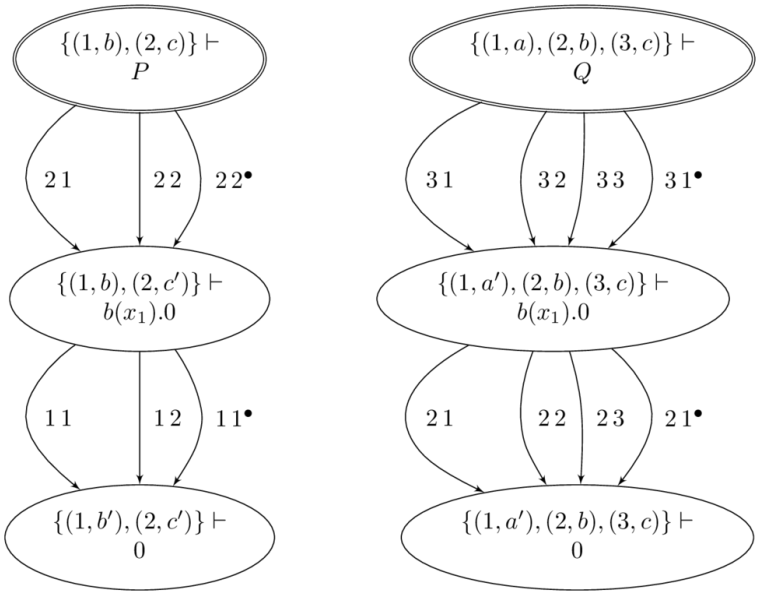

Figure 2.5: Example of n-bisimilar LTSs.

One possible n-bisimulation relation for $n = 3$ is given by:

$$
\begin{aligned}
R = \{&\big(\{(1,b),(2,c)\} \vdash P,\ \{1 \to 2, 2 \to 3\},\ \{(1,a),(2,b),(3,c)\} \vdash Q\big),\\
&\big(\{(1,b),(2,c')\} \vdash b(x_1).0,\ \{1 \to 2, 2 \to 1\},\ \{(1,a'),(2,b),(3,c)\} \vdash b(x_1).0\big),\\
&\big(\{(1,b),(2,c')\} \vdash b(x_1).0,\ \{1 \to 2, 2 \to 3\},\ \{(1,a'),(2,b),(3,c)\} \vdash b(x_1).0\big),\\
&\big(\{(1,b'),(2,c')\} \vdash 0,\ \{1 \to 1, 2 \to 3\},\ \{(1,a'),(2,b),(3,c)\} \vdash 0\big),\\
&\big(\{(1,b'),(2,c')\} \vdash 0,\ \{1 \to 1\},\ \{(1,a'),(2,b),(3,c)\} \vdash 0\big),\\
&\big(\{(1,b'),(2,c')\} \vdash 0,\ \{1 \to 1, 2 \to 1\},\ \{(1,a'),(2,b),(3,c)\} \vdash 0\big),\\
&\big(\{(1,b'),(2,c')\} \vdash 0,\ \{1 \to 2, 2 \to 3\},\ \{(1,a'),(2,b),(3,c)\} \vdash 0\big),\\
&\big(\{(1,b'),(2,c')\} \vdash 0,\ \{1 \to 3, 2 \to 1\},\ \{(1,a'),(2,b),(3,c)\} \vdash 0\big)\}
\end{aligned}
$$

△

2.7 State-of-the-Art

Since the introduction of π-calculus by Milner in the early 1990s, there has been extensive work done investigating the bisimulation in π-calculus and its properties. Naturally, researchers were also highly interested in practical equivalence checking. Due to the previously mentioned challenges of infinite names in π-calculus, the problem was tackled in different ways.

In tools like the Mobility Workbench and Proverif, more restrictive equivalence relations are used. In the Mobility Workbench (MWB)[38], a special variant of bisimulation – open bisimulation – is used for equivalence checking. In MWB, the necessary state space is generated on-the-fly. However, the issue with open bisimulation is that it is more restrictive than early bisimulation and thus results in more false negatives. Similarly, for the applied π-calculus - an extension of π-calculus used for security protocols specification - the tool ProVerif is commonly used, which checks for an equivalence that is stronger than the desired observational equivalence.[5]

The approach taken in this dissertation represents π-calculus via an automaton, which is FRA in this case. There has been a related work done on representing π-calculus as a History-Dependent Automata (HD-Automata)[28]. Similarly to FRAs, it allows for some class of π-calculus systems to be represented by finite-state automata. There has been a tool implemented called HD-Automata Laboratory (HAL)[13]. It works by translating the π-calculus specifications into an HD-automaton and then into an ordinary automaton on which classical bisimulation algorithms and tools can be applied. That is, HAL uses global algorithms for bisimulation checking. Similarly, in work by Mateescu et al.[21],

π-calculus is also translated into another formalism. Moreover, there the acceptable class of π-calculus models is limited to those with finite control, i.e. those without recursive parallel compositions.

There also exists a tool called `PiET` (Pi-Calculus Equivalences Tester) developed by Matteo Mio.[27] The tool allows verifying for ten different types of equivalences, including strong and weak early bisimulations. It implements the ideas developed by Lin[20], and it uses an on-the-fly algorithm. One detail is that it implements support only for guarded sums. It also uses a different definition of weak bisimulation. For example, the following holds in `PiET`: $\overline{a}\langle a\rangle.0 \approx \tau.0+\tau.\overline{a}\langle a\rangle.0$, while it should not hold under the weak bisimulation definition used in this dissertation.

To the best knowledge of the author, there does not exist a publicly available tool or algorithm for checking early bisimulation in π-calculus through the use of FRAs. Additionally, the author could not get access to some potentially equivalent tools such as HAL. Nevertheless, during the work on this dissertation, the author was notified through private communication that a person supervised by Tzevelekos had been independently working on the same problem of bisimulation in π-calculus through FRAs. No details about it are known to the author except that the basis for the approach seems to be similar.

2.8 Closely-related projects

The work in this dissertation is a continuation of the work done by other students under the supervision of V.Koutavas. The high-level end goal of these projects is to allow for equivalence checking of π-calculus models. The two works on which this dissertation builds, in particular, are *Modelling Concurrent Systems: Generation of Labelled Transition Systems of Pi-Calculus Models through the Use of Fresh-Register Automata* by S.Leung[18] and *An Equivalence Checker for Pi-Calculus Models* by B.Contovounesios[8]. The former will be referred to as *Pifra* and the latter as *Pisim* which are the names of the tools produced in the corresponding works. Below, the overview of their works and their relevance to this dissertation is presented.

2.8.1 Pifra

In his dissertation work, Leung tackled the problem of generating finite LTSs for π-calculus models through the use of FRAs. The main contribution of his work is a tool named `pifra`. The tool takes in a π-calculus model and generates an LTS for the equivalent $\times\pi$-calculus model.

One of the essential features of the tool is that it normalises the configurations at each step of its generating algorithm. This is essential for producing finite LTSs. It includes traditional normalisation techniques such as removal of nil processes, removal of unused restrictions, deterministic restriction sorting and scoping, reordering of terms in composition and summation, and normalisation of bound names. Additionally, it applies normalisation to free names to achieve the necessary equivalence of configurations up to a permutation of registers. Fresh names are also normalised, where each fresh name gets the minimum available normalised free name. Notably, the initial normalisation erases the global free names replacing them with normalised names while preserving only the relative alphabetic ordering.

The tool also includes a novel approach to reducing the amount of potentially equivalent states – garbage collection. It achieves the goal by clearing up the registers from names not used in the corresponding process. While this feature is enabled by default, the tool provides an option to turn it off in order to receive a more classical LTS.

Practically, `pifra` is an open-source tool implemented in the Go programming language. The tool accepts a π-calculus model (technically, $\times\pi$-calculus main constructions which are the same as π-calculus) specified in ASCII following the expected grammar. After parsing, the tool stores each process as an abstract syntax tree (AST). This AST is normalised, and the transition rules are applied to generate the final LTS. Internally, the LTS is naturally stored in a graph data structure. When finished, the tool can produce an output in multiple formats such as ASCII or GraphViz DOT file. As a resulting LTS can potentially be infinite, the tool takes in as an argument a limit to the total number of processed states to ensure termination properties.

The final LTS includes all the information, such as the transition labels and state configurations in the form $\sigma \vdash P$. However, all the channel names, free and bound, are normalised. Hence, the original channel name information is lost. Free names normalisation as implemented in `pifra` only keeps the information about the relative alphabetic ordering of the free channel names.

For an example of outputs produced by `pifra`, see how it is used as a part of `pisim` in Examples 2.27, 2.28.

2.8.2 Pisim

This dissertation tackles effectively the same high-level research question as Contovounesios did in his work. That is, an equivalence checker for π-calculus models. His approach was to apply the classical global algorithm - the partition refinement algorithm of Kanellakis and Smolka - directly to the LTS generated by `pifra`. The output of his work was

an open-source tool `pisim` written in Go. The tool takes in two LTSs, merges them, and checks for bisimilarity with the Kanellakis and Smolka algorithm.

The tool implemented by Contovounesios effectively checks for classical strong bisimulation and not for n-bisimulation. The classical strong bisimulation is a stronger equivalence, and thus the equivalence checker is not complete even for finite LTSs as it fails to detect some equivalences. Moreover, the approach is not sound as it can produce false positives. The examples below explain the weaknesses of `pisim`, both examples assume that garbage collection is turned off in `pifra`, but the issues with the approach hold regardless.

Example 2.27 False negative in `pisim`. The tool `pisim` is not complete for finite inputs as it, for example, fails to detect equivalences in presence of register permutation.

Consider the following π-calculus models. Both specified in the syntax accepted by `pifra`. In particular, in this syntax `$s` means νx and `a'<b>` means $\overline{a}\langle b\rangle$. First, $P = c(x).b(x).0$ is given in `jev-a2.1.pi` in Listing 2.1:

```
1 P = c(x).b(x).0
2 P
```

Listing 2.1: File jev-a2.1.pi.

Second, $Q = (\nu x.\overline{x}\langle x\rangle.\overline{a}\langle x\rangle.0) + c(x).b(x).0$ is given in `jev-a2.2.pi` in Listing 2.2:

```
1 Q = ($x.x'<x>.a'<x>.0) + c(x).b(x).0
2 Q
```

Listing 2.2: File jev-a2.2.pi.

From observation, the only difference between the two models is that the latter includes an unreachable choice with an additional free name a. The path to name a is unreachable as an output on a private unshared channel name under a variable x is not possible. Thus, the two models are equivalent.

The produced LTSs can be seen in Figures 2.6 and 2.7. However, `pisim` does not recognise the equivalence as the LTSs produced by `pifra` and consumed by `pisim` have different transitions. This is due to the presence of an additional free name a in `jev-a2.2.pi` which has to be stored in a register in the latter FRA but which is unknown to the former FRA. Secondly, since normalisation preserves the alphabetical order, the free name a ends up in the first register, whereas in the former FRA, the first register contains the normalised name b. As a result, classical bisimulation cannot be applied as the positions of the names in the register set affect the transitions.

△

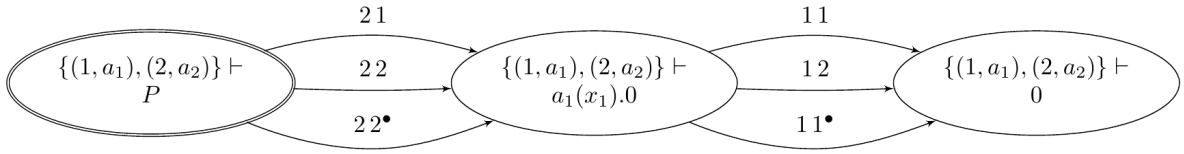

Figure 2.6: The LTS produced for `jev-a2.1.pi`

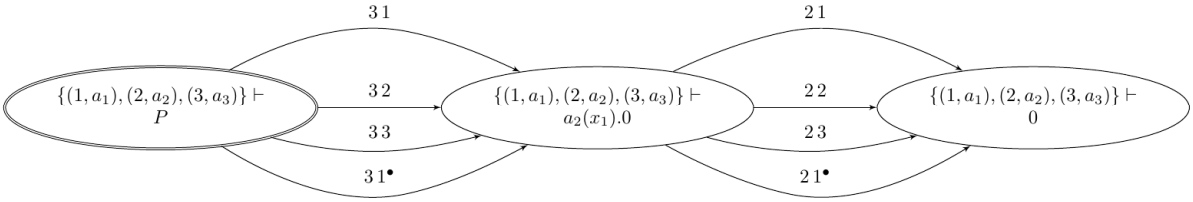

Figure 2.7: The LTS produced for `jev-a2.2.pi`

Example 2.28 False positive in `pisim`. The bisimulation algorithm in `pisim` is not sound for π-calculus bisimulation checking. For example, it does not require a match between the channel names. In particular, `pifra` erases all the information about the original free channel names. Thus, any tool working with the original `pifra` will suffer from the same issue.

Consider the following two π-calculus models. First, `jev-diff-names-1.1.pi` in Listing 2.3:

```
1 b(x).a(x).0
```

Listing 2.3: File jev-diff-names-1.1.pi.

And the second is `jev-diff-names-1.2.pi` in Listing 2.4:

```
1 bb(x).aa(x).0
```

Listing 2.4: File jev-diff-names-1.2.pi.

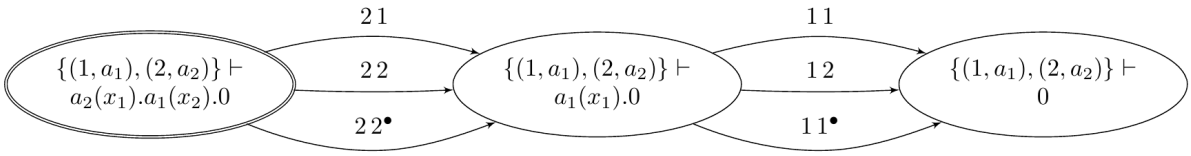

Figure 2.8: `pifra` produces exatcly the same LTS for both `jev-diff-names-1.1.pi` and `jev-diff-names-1.2.pi`

Two models are clearly not equivalent as the free channel names do not match. That is, receiving a name on channel b could not be matched by input on name bb in a bisimulation

game. However, when checked with `pisim`, the output indicates that the models are equivalent. This is because the LTSs generated for these models indeed have equivalent transitions. However, this is not enough for equivalence in π-calculus. The LTSs produced for both models by `pifra` are precisely the same and can be seen in Figure 2.8. Notice that the names are normalised, and since the relative alphabetic order is the same in both models, the names appear the same. Moreover, the original names can be recovered neither from the LTSs nor the `gob` files. △

In conclusion, the implementation of equivalence checking in `pisim` is not satisfactory. Nevertheless, from a practical point of view, there are a few aspects relevant to this dissertation. As mentioned, `pisim` takes in as input two LTSs. To reuse the nice data structures from `pifra`, Contovounesios modified `pifra` to support the output of a serialised internal LTS data structure, which is essentially a graph. For serialisation, the Go programming language's `gob` binary serialisation format was used. Additionally, as mentioned previously, bisimulation is defined for a single LTS, so there is a need to merge two LTSs. In `pisim`, the merge is done explicitly.

Chapter 3

Design

In this chapter, the proposed algorithm for n-bisimulation is presented and explained in detail. It starts with a section explaining Celikkan's algorithm, which is a foundation for the algorithm presented later. The following section then introduces and explains the algorithm for checking n-bisimulation. In the rest of the chapter, arguments for the correctness and complexity of the algorithm are presented.

3.1 Background of the Approach

As mentioned in the preceding chapter (§2.8), this dissertation builds on the works of Leung and Contovounesios. The high-level idea of the solution remains the same as in the original `pisim`. That is, the solution takes two π-calculus models as input. It then runs them through `pifra` to generate corresponding FRAs LTSs. Since bisimulation is defined on the states of a single LTS, the generated LTSs need to be in some way merged. The final step is the currently missing practical algorithm for checking the bisimilarity of states in the merged LTS. However, as is explained later, the original `pifra` also needs modifications to work in this framework.

The proposed algorithm is based on the work of R.Cleaveland and O.Sokosly. In their work [7] they presented a local algorithm, which is attributed to Celikkan, that is capable of determining the bisimilarity of two states of an LTS. Their algorithm can determine a more general parametrised $\langle \Pi, \Phi_1, \Phi_2 \rangle$-bisimulation. For the use case of this dissertation, all Π, Φ_1, and Φ_2 are taken to be universal relations, in which case the algorithm of Celikkan determines classical bisimulation.

The idea behind the algorithm is to construct a directed graph whose vertices are pairs of related states. This is given by the following theorem:

Theorem 3.1. *[7, Theorem 5.1] Let $\langle S, A, \rightarrow \rangle$ be an LTS and $p_0, q_0 \in S$. Then p_0 is*

bisimular to q_0 iff there exists a graph $G = \langle V, E \rangle$ with $V \subseteq S \times S$ and $E \subseteq V \times A \times V$ such that:

- $(p_0, q_0) \in V$;
- *whenever* $(p, q) \in V$ *and* $p \xrightarrow{a} p'$ *then there exists a* q' *such that* $q \xrightarrow{a} q'$ *with* $(p', q') \in V$ *and* $((p', q'), a_1, (p, q)) \in E$;
- *whenever* $(p, q) \in V$ *and* $q \xrightarrow{a} q'$ *then there exists a* p' *such that* $p \xrightarrow{a} p'$ *with* $(p', q') \in V$ *and* $((p', q'), a_2, (p, q)) \in E$;

Intuitively, notice how the conditions in the theorem are the same as those in the self-contained definition of bisimulation given in Definition 2.5. The index i added to actions a in an edge indicates which condition this edge is associated with. Cleaveland and Sokolsky suggest thinking about the incoming edges to a vertex (p, q) as *justification* for including the vertex in the bisimulation.

The exact algorithm is presented below (Algorithms 1, 2, 3, 4) with minor changes: some fixing of, what the author thinks, slight mistakes in the original presentation of the algorithm, and removal of more generic related preorder conditions.

The algorithm consists of three function: $PREORDER$, $SEARCH\_HIGH$ and $SEARCH\_LOW$. Two main global data strictures are used: the graph G and the set of not-related states $\overline{R}$ – both initially empty. Other important data structures are global sets of pointers $high_{p',q,a}$ and $low_{q',p,a}$, and a local set A. The former $high_{p',q,a}$ is used to record the index of the state in an ordered set $\{q' \in S \mid q \xrightarrow{a} q'\}$ that is currently matched to p', and symmetrically for $low_{q',p,a}$. These are needed to ensure that each pair of transitions will be matched at most once and is essential to allow for non-determinism. The set A is used locally within the $PREORDER$ function to record the assumed matchings that need to be re-evaluated. A match between $r \xrightarrow{a} r'$ and $s \xrightarrow{a} s'$ needs to be re-evaluated if the justification (r', s') was found to not be valid.

The execution of the equivalence check for states (p, q) starts with a call to function $PREORDER(p, q)$ (see Algorithm 1). The function returns a value indicating whether the states are related or not, where being related means being equivalent up to currently processed information. Within this function, there are four main steps:

1. [Lines 1-7] The function checks whether the pair of states has been processed before. If a pair has been seen before, then it will either be included as a vertex in the graph G or it will be in the set of not-related states $\overline{R}$. If the pair is in neither of them, the function proceeds by assuming that the states are related and tentatively adding them as a vertex to the graph.

2. [Lines 8-16] For each transition $p \xrightarrow{a} p'$ from p, the function tries to find a matching transition from q. To do that the function calls $SEARCH\_HIGH$ function (Algo-

Algorithm 1: PREORDER(p, q); Part 1/2

Input: p: S, q: S
Output: result : {*related*, *not_related*}

1 **if** $(p, q) \in \overline{R}$ **then**
2 return *not_related*
3 **if** $(p, q) \in V(G)$ **then**
4 return *related*
5 $G := \langle V(G) \cup \{(p, q)\}, E(G)\rangle$
6 $status := related$
7 $A := \{\}$
8 **foreach** $a, p' \in \{a \in A, p' \in S \mid p \xrightarrow{a} p'\}$ **do**
9 **if** $status == not\_related$ **then**
10 break
11 $status := not\_related$
12 **if** $high_{p',q,a}$ *does not exist* **then**
13 create $high_{p',q,a} := 1$
14 $status := SEARCH\_HIGH(p \xrightarrow{a} p', q)$
15 **if** $status == not\_related$ **then**
  /* The block below was modified from the original algortihm */
16 $A := A \cup \{(p, q, r, s, a, i) \mid ((p, q), a_i, (r, s)) \in E(G)\}$
17 **foreach** $a, q' \in \{a \in A, q' \in S \mid q \xrightarrow{a} q'\}$ **do**
18 **if** $status == not\_related$ **then**
19 break
20 $status := not\_related$
21 **if** $low_{q',p,a}$ *does not exist* **then**
22 create $low_{q',p,a} := 1$
23 $status := SEARCH\_LOW(q \xrightarrow{a} q', p)$
24 **if** $status == not\_related$ **then**
25 tccThe block below was modified from the original algortihm
  $A := A \cup \{(p, q, r, s, a, i) \mid ((p, q), a_i, (r, s)) \in E(G)\}$

rithm 3). If $SEARCH\_HIGH$ returns *not_related*, indicating that the transition cannot be matched according to the bisimulation rules, then that indicates that states p and q are also not related. This means that any other pair of states that was justified by the assumed equivalence of p and q needs to be re-evaluated. For that, the set A is filled with corresponding pairs of states.

3. [Lines 17-25] Since the algorithm's goal is bisimulation and not just a simulation, the logic from the previous step is symmetrically applied in the other direction. That is, each transition $q \xrightarrow{a} q'$ from q is attempted to be matched by some $p \xrightarrow{a} p'$

by calling $SEARCH\_LOW$ (Algorithm 4).

4. [Lines 26-41] (See Algorithm 2) The final step is only executed if either in step 2 or 3 it was found that p and q are not equivalent. In this case, vertex (p, q) is removed from G as well as all the edges to and from this vertex. The pair is added to $\overline{R}$. The set A is then iterated to re-evaluate all the matches for which assumptions were violated. The re-evaluation is cascading and reverses the whole chain of now invalid assumptions.

Algorithm 2: PREORDER(p, q); Part 2/2

```
26 if status = not_related then
27     G := ⟨V(G) − {(p,q)}, E(G) − {(π1, a_i, π2) | (π1 = (p,q) ∨ π2 = (p,q))}⟩
28     R̄ := R̄ ∪ {(p,q)}
29     while A! = ∅ do
30         Pop (r, s, r', s', a, type) ∈ A
31         if type == 1 then
32             high_{r',s,a} := high_{r',s,a} + 1
33             status := SEARCH_HIGH(r -a-> r', s)
34         if type == 2 then
35             low_{s',r,a} := low_{s',r,a} + 1
36             status := SEARCH_LOW(s -a-> s', r)
37         if status == not_related then
38             A := A ∪ {(r, s, rr, ss, b, i) | ((r,s), b_i, (rr,ss)) ∈ E(G)}
39             G := ⟨V(G) − {(r,s)}, E(G) − {(π1, b_i, π2) | (π1 = (r,s) ∨ π2 = (r,s))}⟩
40             R̄ := R̄ ∪ {(r,s)}
41 return status
```

Within $SEARCH\_HIGH$ and symmetrically within $SEARCH\_LOW$, the task is – given $p \xrightarrow{a} p'$ and q to find a matching transition $q \xrightarrow{a} q'$, where the equivalence of destination states p', q' is checked by a recursive call to $PREORDER$. If the recursive call returns *related*, meaning that the destination states are deemed to be bisimilar up to the currently processed information, then the relation graph is updated with a new edge from (p', q') to (p, q).

Algorithm 3: SEARCH_HIGH($p \xrightarrow{a} p', q$)

Input: p: S, p': S, q: S, a: A
Output: result : $\{related, not\_related\}$

$status := not\_related$
/* Define a sequence of a-derivatives of q. Assume deterministic order. */
$derivs := \{(ii, q') \in (\mathbb{N}, S) \mid q \xrightarrow{a} q'\}$
while $status == not\_related$ *AND* $high_{p',q,a} \leq |derivs|$ **do**
  $q' := derivs[high_{p',q,a}]$
  $status := PREORDER(p', q')$
  if $status == related$ **then**
    $G := \langle V(G), E(G) \cup \{((p', q'), a_1, (p, q))\}\rangle$
  else
    $high_{p',q,a} := high_{p',q,a} + 1$
return $status$

Algorithm 4: SEARCH_LOW($q \xrightarrow{a} q', p$)

Input: q: S, q': S, p: S, a: A
Output: result : $\{related, not\_related\}$

$status := not\_related$
/* Define a sequence of a-derivatives of p. */
$derivs := \{(ii, p') \in (\mathbb{N}, S) \mid p \xrightarrow{a} p'\}$
while $status == not\_related$ *AND* $low_{q',p,a} \leq |derivs|$ **do**
  $p' := derivs[low_{q',p,a}]$
  $status := PREORDER(p', q')$
  if $status == related$ **then**
    $G := \langle V(G), E(G) \cup \{((p', q'), a_2, (p, q))\}\rangle$
  else
    $low_{q',p,a} := low_{q',p,a} + 1$
return $status$

Example 3.2. Consider the LTS in Example 2.6 with two starting states p_0 and q_0. When executing Celikkan's algorithm on these states, the following graph G_1 is generated:

$$V(G_1) = \{(p_0, q_1), (p_1, q_2), (p_2, q_1)\}$$
$$E(G_1) = \{(p_1, q_2) \xrightarrow{a_1} (p_0, q_1), (p_1, q_2) \xrightarrow{b_1} (p_1, q_2), (p_1, q_2) \xrightarrow{a_1} (p_2, q_1),$$
$$(p_1, q_2) \xrightarrow{a_2} (p_0, q_1), (p_1, q_2) \xrightarrow{b_2} (p_1, q_2), (p_1, q_2) \xrightarrow{a_2} (p_2, q_1)\}$$

△

The correctness of the algortihm is given by Threorem 5.2 in [7], which when adjusted

to the algorithm presented here (restricted to bisimulation) is as follows:

Theorem 3.3. *[7, Theorem 5.2] Let p, q be states in a finite-state LTS, and assume $\overline{R} = \emptyset$ and $G = \langle \emptyset, \emptyset \rangle$. Then $PREORDER(p, q)$ terminates, and the return value of $PREORDER(p, q)$ is related iff $p \sim q$.*

The complexity of the algorithm is given by Threorem 5.3 in [7], which when adjusted is as follows:

Theorem 3.4. *[7, Theorem 5.3] Let $\mathcal{L} = \langle S, A, \rightarrow \rangle$ be an LTS with $|\mathcal{L}| = |S| + |\rightarrow|$. $PREORDER$ takes time proportional to that required by $O(m)$ set membership operations, where $m \leq |\mathcal{L}|^2$.*

3.2 The algorithm

In this section the algorithm for checking n-bisimulation is presented. The algorithm is an extension to the Celikkan's algorithm presented earlier in this chapter. Therefore, the algorithm presented is a local one.

The higher level goal of the algorithm is to check for equivalence between two $\times\pi$-calculus models. At a lower level the algorithm deals with checking for n-bisimulation between two states in an $\times\pi$-calculus LTS. This LTS being checked is a result of implicitly merging the two LTSs corresponding to the original $\times\pi$-calculus models.

3.2.1 Preliminaries

As described in the background, the semantics of a $\times\pi$-calculus model is described by an LTS where states are taken from $O(K)$, labels are as given in Definition 2.19, and transition relation is as given in Table 2.2. Note that LTS is restricted to double-label transitions. The original algorithm of Celikkan checks for classical bisimulation which for the described LTS would be a relation $R_{cb} \subseteq O(K) \times O(K)$. However, as explained on the example of the original `pisim` tool, this type of bisimulation is not sufficient. Thus, n-bisimulation is needed which is a relation $R_n \subseteq O(K) \times ([n] \rightleftharpoons [n]) \times O(K)$ for which the n-bisimulation rules hold.

To adapt Celikkan's algorithm to n-bisimulation, let νR_n be a new relation stemming from R_n such that $\nu R_n \subseteq (O(K) \times ([n] \rightleftharpoons [n])) \times (O(K) \times ([n] \rightleftharpoons [n]))$ and the following property always holds – if $((\sigma_1 \vdash P_1, \rho_1), (\sigma_2 \vdash P_2, \rho_2)) \in \nu R_n$, then $\rho_1 = \rho_2^{-1}$ – that is, the mappings are inverses of each other. Each relation νR_n is then said to have a corresponding parent relation R_n. Two relations are related with the following definition:

Definition 3.5. $(\sigma_1 \vdash P_1, \rho, \sigma_2 \vdash P_2) \in R_n$ iff $((\sigma_1 \vdash P_1, \rho), (\sigma_2 \vdash P_2, \rho^{-1})) \in \nu R_n$ $\Diamond$

A new notion of bisimilarity can be defined for relations νR_n.

Definition 3.6. A relation νR_n is said to be ν_n-bisimilar iff a corresponding parent relation R_n is n-bisimilar. $\Diamond$

Hence, given these formalisms, n-bisimulation can be checked through a ν_n-bisimulation. This new relation νR_n has the form of classical bisimulation, and thus some classical algorithm such as that of Celikkan can be adapted.

Note that a local algorithm was chosen here, even though a variant of a partition algorithm can potentially also be applied on an LTS with states in $O(K) \times ([n] \rightleftharpoons [n])$. It is not clear how the original LTS of two $\times\pi$-calculus models can be transformed into the new form efficiently. The presented local algorithm demonstrates one possible way to build this LTS on-the-fly while simultaneously trying to build the equivalence relation.

3.2.2 Preprocessing

Before running the bisimulation algorithm, some preprocessing needs to be done to obtain the starting states of the form $O(K) \times ([n] \rightleftharpoons [n])$. All the states of this form will be referred to as ν-states. The input to the preprocessing stage are two LTSs of the two corresponding $\times\pi$-calculus models. Within each LTS one state corresponding to the top level process is designated as a starting state. The preprocessor then constructs the starting ν-states. The initial mapping ρ_0 is obtained from the definition of n-bisimulation, which states that the initial register set in the starting state only contains all the free names within the starting process, and the initial mapping ρ_0 is a mapping matching all the same names between two starting register sets. This initial mapping can be constructed deterministically by simply matching the same names. This is deterministic since, by the definition of FRA and $\times\pi$-calculus used here, a name can appear at most once in a given register set.

3.2.3 The algorithm

In the algorithm, ν-states will be prefixed with n, e.g. nP denotes a ν-state, while p denotes a regular state.

The algorithm assumes the following globally available data structures:

- Two LTSs L_1, L_2 corresponding to the input $\times\pi$-calculus models. These LTSs are given by $\langle S, A, \rightarrow, s_0 \rangle$ where $S \subseteq O(K)$ and $s_0 \in S$ is the starting state. These LTSs are referred to as original LTSs. The algorithm uses these original LTSs to on-the-fly build a new LTS with states in $O(K) \times ([n] \rightleftharpoons [n])$ for calculating n-bisimulation.

- Graph G – it is similar to the graph in the Celikkan's algorithm. The only difference is the exact types of vertices and edges.
- A set $\overline{R}$ of not-related states.
- Sets of pointers $high$, low, $highK$, $lowK$. The former $high$ and low are logically equivalent to their counterparts in Celikkan's algorithm. The latter $highK$, $lowK$ are extensions of the former with additional parameter added for the value of k'. Overall, these sets are used to ensure that every two transitions are matched at most once. All the pointers are assumed to be mapped to 0 by default.
- A map M from ν-states to the original states. It represents a relation between ν-states and so-called originating states. Each ν-state nP is constructed based on some state p from the original LTS. This is recorded with $M[nP] = p$.

Additionally, similarly to Celikkan's algorithm, a local set A is used to track matches needing re-evaluation.

The n-bisimulation algorithm is defined in terms of three main functions: $BISIM$, $MATCH\_LEFT$ and $MATCH\_RIGHT$. The algortihm is presented in Algortihms 5, 6 for $BISIM$, and in Algorithms 7, 8 for $MATCH\_LEFT$. The description of $MATCH\_RIGHT$ is symmetrical to that of $MATCH\_LEFT$ and is omitted.

The entry point of the algorithm is a function $BISIM$, which is a modified version of $PREORDER$ from Celikkan's algorithm. Within it, calls are made to $MATCH\_LEFT$ and $MATCH\_RIGHT$, which are similar to $SEARCH\_HIGH$ and $SEARCH\_LOW$ in Celikkan's algorithm.

The key difference between $BISIM$ and $PREORDER$ is the type of the states and how the iteration over transitions is performed. In $BISIM$, the main units of work are two ν-states nP and nQ. However, in this function, the mappings ρ are not explicitly utilised, making it look very similar to the original $PREORDER$. The most important aspect is how iteration is done in two main loops that match the transitions in both directions. Each ν-state has a so-called originating classical state from the original LTS with the mappings between the two given by M. Since the transitions between the ν-states are unknown, as otherwise a partition algorithm could be used, the iterations proceeds on the transitions from the originating state. E.g. Consider the first for-loop (lines 9-15 in Algorithm 5). There the iteration is done on transitions $p \xrightarrow{a} p'$, where p is the originating state of ν-state nP. For each transition, the function calls $MATCH\_LEFT$ to seek a matching transition from nQ based on its respective originating state q.

Otherwise, the logic in $BISIM$ is mostly the same as in $PREORDER$. One interesting aspect is that edges in the graph here contain additional information about k' and v, where first is needed to deal with the $FINP.2$ rule, and the latter is needed to be able

Algorithm 5: BISIM(nP, nQ); Part 1/2

Input: nP: $O(K) \times ([n] \rightleftharpoons [n])$, nQ: $O(K) \times ([n] \rightleftharpoons [n])$
Output: result : $\{related, not\_related\}$

```
1  if (nP, nQ) ∈ R̄ then
2      return not_related
3  if (nP, nQ) ∈ V(G) then
4      return related
5  G := ⟨V(G) ∪ {(nP, nQ)}, E(G)⟩
6  status := related
7  A := {}
8  p := M[nP]; q := M[nQ]
9  foreach a, p' ∈ {a ∈ A, p' ∈ S | p -a-> p'} do
10     if status == not_related then
11         break
12     status := not_related
13     status := MATCH_LEFT(nP, nQ, p', a)
14     if status == not_related then
15         A := A ∪ {(nR, nS, a, i, k', v) | ((nP, nQ), a_i, k', v, (nR, nS)) ∈ E(G)}
16 foreach a, q' ∈ {a ∈ A, q' ∈ S | q -a-> q'} do
17     if status == not_related then
18         break
19     status := not_related
20     status := MATCH_RIGHT(nQ, nP, q', a)
21     if status == not_related then
22         A := A ∪ {(nR, nS, a, i, k', v) | ((nP, nQ), a_i, k', v, (nR, nS)) ∈ E(G)}
```

to re-evaluate the transitions given the previously mentioned specifics of iteration using originating states. Similarly, the data structures stored in local set A are expanded to contain all the necessary information needed for re-evaluation. Additionally, re-evaluation logic is extended to cover the $FINP$ rule.

Applying n-bisimulation rules

The key changes to the Celikkan's algorithm are how the transitions between the ν-states are created and matched. These are reflected in $MATCH\_LEFT/RIGHT$ functions. The challenge when working with n-bisimulation is that the transition labels do not need to match exactly. Instead, they need to match according to the $NSYM$ rules.

The approach taken in the presented algorithm is a forward direction one, where a matching label is constructed using the rules, and then a transition with this label is

Algorithm 6: BISIM(nP, nQ); Part 2/2

```
23 if status = not_related then
24     G := ⟨V(G) − {(nP, nQ)}, E(G) − {(π1, ai, k′, v, π2) | (π1 = (nP, nQ) ∨ π2 =
       (nP, nQ))}⟩
25     R̄ := R̄ ∪ {(nP, nQ)}
26     while A ≠ ∅ do
27         Pop (nR, nS, a, type, k′, u, v) ∈ A
28         if type == 1 then
29             if k′ is defined then
30                 highK[nR, nS, v, a, k′] := highK[nR, nS, v, a, k′] + 1
31             else
32                 high[nR, nS, v, a] := high[nR, nS, v, a] + 1
33             status := MATCH_LEFT(nR, nS, v, a)
34         if type == 2 then
35             if k′ is defined then
36                 lowK[nS, nR, v, a, k′] := lowK[nS, nR, v, a, k′] + 1
37             else
38                 low[nS, nR, v, a] := low[nS, nR, v, a] + 1
39             status := MATCH_RIGHT(nS, nR, v, a)
40         if status == not_related then
41             A := A ∪ {(nRR, nSS, b, i, k″, v′) | ((nR, nS), bi, k″, v′, (nRR, nSS)) ∈
               E(G)}
42             G := ⟨V(G) − {(nR, nS)}, E(G) − {(π1, bi, k″, v′, π2) | (π1 =
               (nR, nS) ∨ π2 = (nR, nS))}⟩
43             R̄ := R̄ ∪ {(nR, nS)}
44 return status
```

searched for. For each transition, $p \xrightarrow{a} p'$ from the originating state p, the algorithm classifies the transition into one of the $NSYM$ rules based on the label and the mapping ρ of the nP state. By following the $NSYM$ rules directly, the algorithm can construct a matching label a' and then build a sequence of a'-derivatives of the matched originating state q, that is, all q' for which $q \xrightarrow{a'} q'$. At this stage, there is enough information to construct the destination ν-states nPX and nQX with the potentially updated register mappings. That is, transition $nP \xrightarrow{a} nPX$ is matched with transition $nQ \xrightarrow{a'} nQX$. For all the rules other than $FINP$, the rest of the logic follows Celikkan's algorithm. The potential match is checked via a recursive call to $BISIM$, and if determined to be related, the graph gets updated with a new edge. The part for these rules in the left-to-right direction can be seen in Algorithm 7.

An exception is the handling of the $FINP$ rule. Unlike the other rules, $FINP$ consists

Algorithm 7: MATCH_LEFT(nP, nQ, p', a); PART 1/2

Input: nP: $O(K) \times ([n] \rightleftharpoons [n])$, nQ: $O(K) \times ([n] \rightleftharpoons [n])$, p': $O(K)$, a: A
Output: result : $\{related, not\_related\}$

$status := not\_related$
$p := M[nP]$; $q := M[nQ]$
/* If not FINP rule */
if $a \neq ij^{\bullet}$ **then**
  if $a == \tau$ **then**
    /* TAU-rule */
    /* Define a sequence of τ-derivatives of q. */
    $derivs := \{(ii, (q', \emptyset, \emptyset)) \in (\mathbb{N}, (O(K), [n], [n])) \mid q \xrightarrow{\tau} q'\}$
  else if $a == ij \ AND \ j \in dom(nP.Rho)$ **then**
    /* INP1 */
    $\alpha_1 := np.Rho[i] \ np.Rho[j]$
    $derivs := \{(ii, (q', j, j)) \in (\mathbb{N}, (O(K), [n], [n])) \mid q \xrightarrow{\alpha_1} q'\}$
  else if $a == ij \ AND \ j \notin dom(nP.Rho)$ **then**
    /* INP2 */
    /* Here $k^{\bullet}$ is a quantifier over $[n]$ */
    $derivs := \{(ii, (q', j, k)) \in (\mathbb{N}, (O(K), [n], [n])) \mid q \xrightarrow{nPX.Rho[i]k^{\bullet}} q'\}$
  else if $a == \overline{ij} \ AND \ j \in dom(nP.Rho)$ **then**
    /* OUT */
    $\alpha_2 := \overline{np.Rho[i]} \ np.Rho[j]$
    $derivs := \{(ii, (q', j, j)) \in (\mathbb{N}, (O(K), [n], [n])) \mid q \xrightarrow{\alpha_2} q'\}$
  else if $a == \overline{ij}^{\circledast}$ **then**
    /* FOUT */
    $derivs := \{(ii, (q', j, k)) \in (\mathbb{N}, (O(K), [n], [n])) \mid .q \xrightarrow{nPX.Rho[i]k^{\circledast}} q'\}$
  while $status == not\_related \ AND \ high[nP, nQ, p', a] \leq |derivs|$ **do**
    $q', j, k := derivs[high[nP, nQ, p', a]]$
    $status, nPX, nQX := MATCH\_LEFT\_HELP(nP, nQ, p', q', j, k, a)$
    if $status == related$ **then**
      $G := \langle V(G), E(G) \cup \{((nPX, nQX), a_1, \emptyset, p', (nP, nQ))\}\rangle$
    else
      $high[nP, nQ, p', a] += 1$

of two parts: $FINP.1$, which is similar to $INP2$, and $FINP.2$, which is unique in that it potentially matches one left transition to multiple right transitions. For the rule to hold, both parts need to hold, so multiple matches need to hold. Note that non-determinism is allowed, so all possible combinations of acceptable matches might need to be considered. The part for this rule can be seen in Algorithm 8. The logic for $FINP.1$ directly follows that for $INP2$ with the only difference in that the new edge is not directly added to the graph, but instead, it is added to a temporary set. The logic for $FINP.2$ involves building

Algorithm 8: MATCH_LEFT(nP, nQ, p', a); PART 2/2

```
   /* FINP                                                              */
23 else if a == ij• then
       /* FINP.1                                                        */
24     derivs := {(ii, (q', j, k)) ∈ (ℕ, (O(K), [n], [n])) | q --nPX.Rho[i]k•--> q'}
25     edges := {}
26     while status == not_related AND high[nP, nQ, p', a] ≤ |derivs| do
27         q', j, k := derivs[high[nP, nQ, p', a]]
28         status, nPX, nQX := MATCH_LEFT_HELP(nP, nQ, p', q', j, k, a)
29         if status == related then
               /* Do not add edges immediately, wait for FINP.2         */
30             edges := edges ∪ {((nPX, nQX), a1, ∅, p', (nP, nQ))}
31         else
32             high[nP, nQ, p', a]+ = 1
       /* FINP.2                                                        */
33     ((σ2, Q), ρnQ) := nQ
34     for k' in dom(σ2) − img(ρnQ) do
35         α3 := nP.Rho[i] k'
36         derivs' := {(ii, (q', j, k')) ∈ (ℕ, (O(K), [n], [n])) | q --α3--> q'}
37         while status == not_related AND highK[nP, nQ, p', a, k'] ≤ |derivs'| do
38             q', j, k' := derivs'[highK[nP, nQ, p', a, k']]
39             status, nPX, nQX := MATCH_LEFT_HELP(nP, nQ, p', q', j, k', a)
40             if status == related then
41                 edges := edges ∪ {((nPX, nQX), a1, k', p', (nP, nQ))}
42             else
43                 highK[nP, nQ, p', a, k']+ = 1
44     if status == related then
45         G := ⟨V(G), E(G) ∪ edges⟩
46 return status
```

a set of all possible k' and then following the logic for $INP1$, but, again, not adding the edges immediately. An important detail in handling $FINP.2$ is that the new edges in the dependency graph also include the value of k'. This is needed to be able to trigger a re-evaluation of $FINP.2$. Specifically, as it uses a different set of counters $highK$, which has k' as a parameter.

Example 3.7. For visualisation of how ν-states are related to originating states, see Figure 3.1. Dotted lines correspond to the mapping M. △

The main functions described above use two helper functions: $MATCH\_LEFT\_HELP$ and FIX_GC. The functions can be found in Algorithms

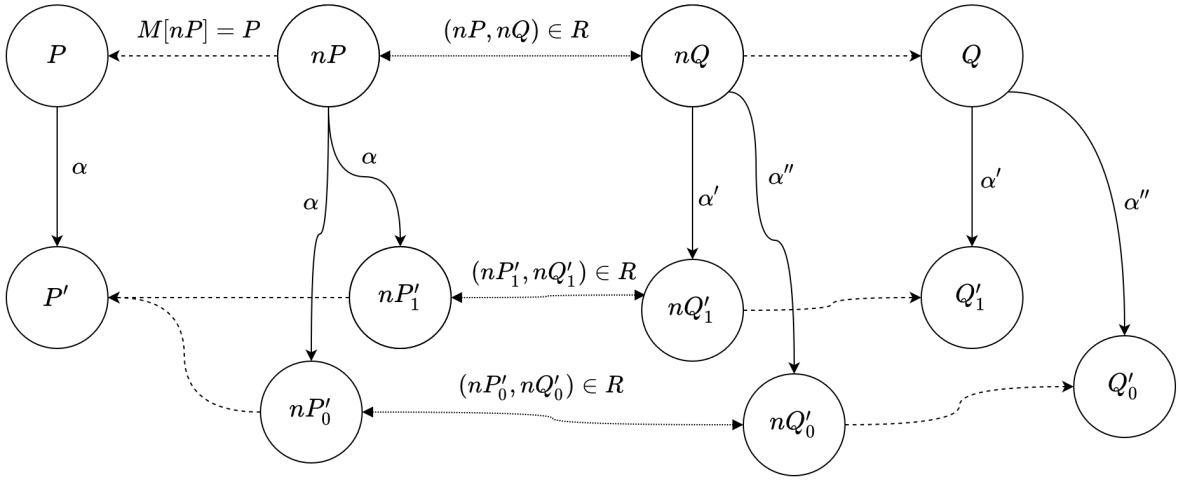

Figure 3.1: Example of the relation between originating and ν-states. Dotted lines correspond to the mapping M.

9 and 10. The former contains the common logic for all the rules, where destination ν-states are constructed with appropriate register mappings as per the rules, and a call to $BISIM$ is made to check the relation between the candidate matched states. The latter FIX_GC deals with garbage collection as defined by Leung and as implemented in `pifra`. As garbage collection can remove some no-longer-necessary names from the registers, some adjustments are needed to the mapping ρ between the register sets. All the mappings from and to removed names are also removed from ρ' to keep the mapping valid.

Algorithm 9: MATCH_LEFT_HELP(nP, nQ, p', q', j, k, a)

Input: nP, nQ, p': $O(K)$, q': $O(K)$, j: $\mathbb{N}$, k: $\mathbb{N}$, a: A
Output: result : $\{related, not\_related\}$, nPX, nQX

```
1  status := not_related
2  ρ' := nP.Rho
3  if a ≠ τ then
4      ρ' := ρ'[j ↔ k]
5  nPX := (p', ρ')
6  nQX := NIL
7  if ρ'^{-1} exists then
8      nQX := (q', ρ'^{-1})
9      if garbage collection is enabled then
10         nPX, nQX := GC_FIX(nPX, nQX)
11     M[nPX] := p' ; M[nQX] := q'
12     status := BISIM(nPX, nQX)
13 return status, nPX, nQX
```

Algorithm 10: GC_FIX(nPX, nQX)

Input: nPX: $O(K) \times ([n] \rightleftharpoons [n])$, nQX: $O(K) \times ([n] \rightleftharpoons [n])$
Output: nPX, nQX

1 $p' := M[nPX]$
2 $q' := M[nQX]$
3 $(\sigma_1, P') := p'$
4 $(\sigma_2, Q') := q'$
5 $nPX.Rho := nPX.Rho \cap \{(i,j) \mid (i,j) \in nPX.Rho \wedge \sigma_1(i) \in \mathbb{A} \wedge \sigma_2(j) \in \mathbb{A}\}$
6 $nQX := (q', nPX.Rho^{-1})$
7 return nPX, nQX

Example 3.8. To better understand the algorithm presented, consider the models from Example 2.26. There are two equivalent processes P and Q given by:

$$P = c(x).b(x).0$$
$$Q = (\nu x.\overline{x}\langle x\rangle.\overline{a}\langle x\rangle.0) + c(x).b(x).0$$

The first step is generating symbolic LTSs for these processes. This is achieved by using `pifra`, and the LTS are visualised in Figure 2.5.

The starting states are $\{(1,b),(2,c)\} \vdash P$ and $\{(1,a),(2,b),(3,c)\} \vdash Q$. As described, preprocessing is needed to generate the initial mapping between the registers. This mapping is $\rho = \{1 \to 2, 2 \to 3\}$. There is now enough information to produce two starting ν-states:

$$nP1 = \big(\{(1,b),(2,c)\} \vdash P, \{1 \to 2, 2 \to 3\}\big)$$
$$nQ1 = \big(\{(1,a),(2,b),(3,c)\} \vdash Q, \{2 \to 1, 3 \to 2\}\big)$$

The checking for strong n-bisimulation is initiated by making a call to $BISIM(nP1, nQ1)$. Since this is the first call, the pair of states is neither in $\overline{R}$ nor in $V(G)$. Thus, it is tentatively added to $V(G)$, and the matching starts. Recall that the transitions from the original LTSs are used for matching. For $nP1$ the originating state is $\{(1,b),(2,c)\} \vdash P$. Hence, for the left-to-right direction, all the transitions from $\{(1,b),(2,c)\} \vdash P$ are considered and matched to transitions from $\{(1,a),(2,b),(3,c)\} \vdash Q$ based on the $NSYM$ rules and register mapping ρ.

For a more interesting scenario, consider the transition $\{(1,b),(2,c)\} \vdash P \xrightarrow{22^\bullet} \{(1,b),(2,c')\} \vdash b(x_1)$. Within $MATCH\_LEFT$ this transition is classified as needing $FINP$ rule. Within $FINP.1$ part, the sequence *derivs* would only contain a single transition $\{(1,a),(2,b),(3,c)\} \vdash Q \xrightarrow{31^\bullet} \{(1,a'),(2,b),(3,c)\} \vdash b(x_1).0$.

Using the two matched transitions, new ν-states nPX, nPQ can be created, where:

$$nPX = \big(\{(1,b),(2,c')\} \vdash b(x_1), \{1 \to 2, 2 \to 1\}\big)$$
$$nQX = \big(\{(1,a'),(2,b),(3,c)\} \vdash b(x_1).0, \{2 \to 1, 1 \to 2\}\big)$$

A recursive call to $BISIM(nPX, nQX)$ is made, and since the states are indeed bisimilar, the call would succeed.

Since $FINP$ is a composite rule, after $BISIM(nPX, nQX)$ successfully returns, the second part $FINP.2$ is executed. There is only one $k' = 1$ to consider and only one transition $\{(1,a),(2,b),(3,c)\} \vdash Q \xrightarrow{31} \{(1,a'),(2,b),(3,c)\} \vdash b(x_1).0$. Coincidently, the new ν-states generated are the same as in $FINP.1$. Thus, a call to $BISIM(nPX, nQX)$ will short circuit since $(nPX, nQX) \in V(G)$ from the previous call.

In conclusion, it can be observed that the proposed algorithm directly follows the definition of n-bisimulation. It naturally builds on top of Celikkan's algorithm to produce the required results. △

3.3 Correctness

In this section, an argument for the correctness of the algorithm is presented. The idea for the arguments follows from Celikkan's algorithm.

The following lemma formalises the connection between ν_n-bisimulation and n-bisimulation.

Lemma 3.9. Given two $\times\pi$-calculus processes P and Q, if $((\sigma_{01}) \vdash P, \rho), (\sigma_{02} \vdash Q, \rho^{-1}) \in \nu R_n$, where νR_n is ν_n-bisimulation, $\rho = \sigma_{01} \leftrightarrow \sigma_{02}$ for some σ_{01}, σ_{02} with $img(\sigma_{01}) = fn(P_1)$ and $img(\sigma_{02}) = fn(P_2)$, then P and Q are n-bisimilar. ♡

Proof sketch. The proof follows directly from the definitions of ν_n-bisimulation and n-bisimulation. □

Similarly to Cellikan's algorithm, the key idea is in constructing a directed graph G, which, when necessary conditions are satisfied, represents an n-bisimulation and thus ν_n-bisimulation. Consider this slightly informal theorem:

Theorem 3.10. *Let $\langle S, A, \to\rangle$ be an LTS and $p_0, q_0 \in S$ such that $S \subseteq O(K) \times ([n] \rightleftharpoons [n])$. Then p_0 is ν_n-bisimilar to q_0 iff there exists a graph $G = \langle V, E\rangle$ with $V \subseteq S \times S$ and $E \subseteq V \times A \times \mathbb{N} \times V$ such that:*

- $(p_0, q_0) \in V$;

- *whenever* $(p, q) \in V$ *and* $p \xrightarrow{a} p'$ *then there exists* q' *such that* $q \xrightarrow{a'} q'$ *with* $(p', q') \in V$ *and* $((p', q'), (a_1, k'), (p, q)) \in E$. *And the relationship between* p' *and* q' *as well as between* a *and* a' *satisfy the* $NSYM$ *rules in a natural way;*
- *whenever* $(p, q) \in V$ *and* $q \xrightarrow{a} q'$ *then there exists* p' *such that* $p \xrightarrow{a'} qp$ *with* $(p', q') \in V$ *and* $((p', q'), (a_2, k'), (p, q)) \in E$. *And the relationship between* p' *and* q' *as well as between* a *and* a' *satisfy the* $NSYM$ *rules in a natural way.*

Proof sketch. The proof would be similar to Theorem 3.1. Notice that the definition of the graph follows precisely the definition of the n-bisimulation. □

The following theorem then establishes the correctness of the proposed algorithm.

Theorem 3.11. *Let* $p, q \in O(K)$ *be states in a finite-state LTS corresponding to two* $\times\pi$*-calculus models. Let* n *be the size of the register set, and let* ρ_0 *be the initial mapping between the names in registers of* p *and* q *as required by* n*-bisimulation. Let* $nP = (p, \rho_0)$ *and* $nQ = (q, \rho_0^{-1})$*, and assume* $\overline{R} = \emptyset$ *and* $G = \langle \emptyset, \emptyset \rangle$. *Then* $BISIM(nP, nQ)$ *terminates, and the return value of* $BISIM(nP, nQ)$ *is "related" iff* p *is* n*-bisimilar to* q.

Proof. First, consider termination. Notice that since the input LTS is finite, the number of possible vertices and edges in G is also finite. $BISIM$ can fully execute only once per each state pair, as each pair after a call to $BISIM$ is either a vertex in a graph G or is in a set $\overline{R}$. Each vertex is added or removed from the graph G at most once, and it is added to $\overline{R}$ also at most once. This bounds the number of calls to $MATCH\_LEFT$ and $MATCH\_RIGHT$ in the two matching loops.

Notice that each edge is added or removed from the graph G at most once. A new edge can only be added once in $MATCH\_LEFT/RIGHT$, and exclusivity is guaranteed by the monotonically increasing counters $high$, low, $highK$, $lowK$. Meanwhile, across the re-evaluation loops, the number of calls to $MATCH\_LEFT/RIGHT$ is bounded because there is a finite number of possible edges in $E(G)$ and each edge can only be re-evaluated a finite number of times. Each re-evaluation necessarily increases one of the counters. And a counter can only increase a finite number of times until an edge is eventually removed from $E(G)$.

Second, the correctness. The invariant is that whenever $BISIM$ returns, the graph G satisfies the conditions in Theorem 3.10 up to the LTS generated and processed so far. Execution of $BISIM$ terminates if either the whole LTS has been processed and a bisimulation was found or if one of the conditions could not be satisfied, thus making it unnecessary to proceed further. In the former case, *related* is returned, meaning that the invariant holds for the entire LTS; thus, the starting states are n-bisimilar. In the latter case, *not_related* is returned as states are not n-bisimilar due to a counter-example found.

The connection between ν_n-bisimulation and n-bisimulation comes from the preceding lemma. □

3.4 Complexity

In this section, the time complexity of the proposed algorithm is explored.

Theorem 3.12. *Let $\mathcal{L}_1 = \langle S_1, A, \rightarrow_1 \rangle$ and $\mathcal{L}_2 = \langle S_2, A, \rightarrow_2 \rangle$ be two LTSs, one per each input $\times\pi$-calculus model. Let n be the parameter of n-bisimulation, corresponding to the size of register sets. Then $BISIM$ has a time complexity in $O(V * | \rightarrow_1 | * | \rightarrow_2 |)$ with $V = |S_1| * |S_2| * 2^n * 2^n * n!$ when measured in terms of set membership operations and when garbage collection is not applied.*

Proof. The input to the algorithm are two LTSs: $\mathcal{L}_1 = \langle S_1, A, \rightarrow_1 \rangle$ and $\mathcal{L}_2 = \langle S_2, A, \rightarrow_2 \rangle$, where S_1 and S_2 are both subsets of $O(K)$. Two LTSs are merged implicitly. However, the algorithm technically operates on an expanded LTS $\mathcal{L}' = \langle S', A, \rightarrow' \rangle$, where $S' \subseteq O(K) \times ([n] \rightleftharpoons [n])$. This LTS is built on-the-fly. The preprocessing needed to build two initial states in $\mathcal{L}'$ from starting states in $\mathcal{L}_1$ and $\mathcal{L}_2$ is in $O(n)$ as free names simply need to be matched to construct the initial mapping ρ_0.

There are four types of two-step transitions as given by Definition 2.19. Unlike in the regular bisimulation, in n-bisimulation, the matched labels do not need to be the same, and a label α can, for some rules, be matched to one of α_1 or α_2 such that $\alpha_1 \neq \alpha_2$. For example, label 12 can, under certain circumstances, be matched to either $12^\bullet$ or $11^\bullet$. Overall, there are $4n^2 + 1$ unique transition labels.

Consider the state expansion when going from regular states to ν-states. Due to the presence of partial bijections in ν-states, the problem is susceptible to combinatorial explosion. For an LTS with register sets of size n, there are $pb\_num = \sum_{i=0}^{n} \left(\binom{n}{i} * \frac{n!}{(n-i)!} \right) = \sum_{i=0}^{n} \left(\binom{n}{i} * \binom{n}{i} * i! \right)$ unique partial bijections. The maximum number of unique vertices in graph G can then given by $V = |S_1| * |S_2| * pb\_num$.

Ignoring the recursive calls, consider the complexity of executing $BISIM$ for a pair (nP, nQ). Three main parts include iteration. There is a loop that matches each transition from nP to some transition from nQ, which will be called a p-loop, there is a symmetric loop in the other direction from nQ to nP called q-loop, and there is a re-evaluation loop iterating over the set A. The complexity of p-loop is in $O(| \rightarrow_1 | * t(MATCH\_LEFT))$. Note that iteration is done over transitions from an originating state p and not from the ν-state nP. Thus, the number of transitions is in $O(| \rightarrow_1 |)$. Within $MATCH\_LEFT$, the iteration is done over the *derivs* sequence, the complexity of $MATCH\_LEFT$ for all the rules is in $O(| \rightarrow_2 |)$. Hence, since $BISIM$ can only be fully executed once per each

vertex, the total complexity for executing p and q loops is in $O(V * | \rightarrow_1 | * | \rightarrow_2 |)$. In particular, total number of transitions can be bounded by $| \rightarrow_i | \leq |S_i|^2 * n^2$. Hence, the previous complexity is also in $O(V * |S_1|^2 * |S_2|^2 * n^4)$, which is also in $O(V^3)$.

As for the re-evaluation loop, if a pair (nP, nQ) is found to be *not_related*, then all the other pairs dependent on it need to be re-evaluated. In the worst case, all the edges need to be re-evaluated. This means the total complexity of all re-evaluations is in $O(V * | \rightarrow_1 | * | \rightarrow_2 |)$, the same as for the q and p loops. □

The previous theorem assumed that garbage collection was not applied. However, the algorithm does have support for garbage collection. That is, it can work on LTSs that follow the property of garbage collection as defined by Leung in [18, Lemma 5.3]. This means support for the erasure of registers across transitions. To work with garbage collection, additional logic is required and is presented in Algorithm 10. While the reduction in the number of states in an LTS resulting from garbage collection is hard to quantify, additional runtime overhead is given by the following theorem:

Theorem 3.13. *To correctly work on LTSs generated with garbage collection, $BISIM$ requires an additional time overhead per each potential matching with complexity in $O(n)$ in terms of set membership, where n is the size of the registers set in n-bisimulation.*

Proof sketch. When working with garbage collected LTSs, GC_FIX has to run for every potential match. The runtime complexity is in $O(n)$ when measured in terms of set membership operations. This is evident from the definition of GC_FIX, where each entry in the register mapping ρ needs to be checked. □

The worst-case time complexity of $BISIM$ looks intimidating due to the presence of the factorial of n. However, it is unclear how the adversarial pattern can be produced from $\times\pi$-calculus models. The author was not able to find $\times\pi$-calculus models demonstrating factorial or exponential growth in terms of n.

The factorial term comes from the total number of partial bijections. Three rules can potentially create new bijections: $INP2$, $FOUT$, and $FINP$. The last rule is of particular interest as $FINP.2$ is the only rule where a single transition might need to be matched with multiple other transitions. An ideal worst-case scenario would have multiple levels where a single $ij^\bullet$ transition is being matched with n other transitions of the form ik', producing n new mappings $\rho[i \rightarrow k']$. Consider a hypothetical Example 3.14.

Example 3.14 Generating all partial bijections. Consider the schematic of an LTS in Figure 3.2. There are two systems in the LTS, and the goal is to check if $P1$ is equivalent to $Q1$. There are just 3 transitions on the left side, and $O(n^2)$ transitions on the right

side, where n is the size of the register set. Assume that the starting ρ is $\{1 \to 1\}$, and that $P1$ has 1 name in the registers, while $Q1$ has n names in the registers. The starting pair is then $((P1, \{1 \to 1\}), (Q1, \{1 \to 1\}))$.

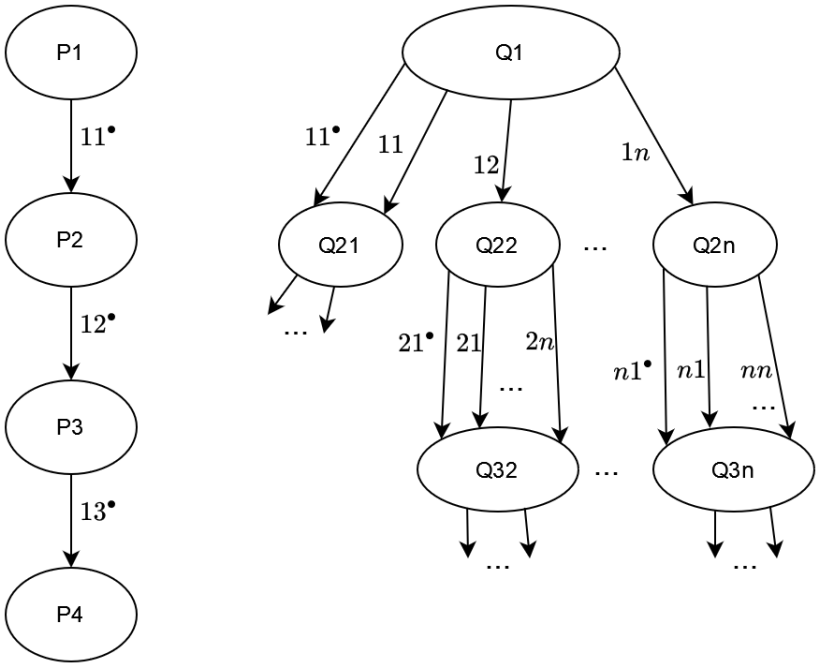

Figure 3.2: Example of a hypothetical LTS generating all partial bijections.

When playing the bisimulation game from left states to right states, for a fresh transition $P1 \xrightarrow{11^\bullet} P2$ the potential matches for state $P2$ are the states $Q21, Q22, ..., Q2n$. To verify the matches conditions of the $FINP$ rule need to be checked. In particular, $FINP.1$ matches $11^\bullet$ with $11^\bullet$. While $FINP.2$ rule says that all the transitions of the form $Q1 \xrightarrow{1k'} Q2k'$ for $k' \in [n] \setminus \{1\}$ need to be checked. From the figure it is evident that there are $n-1$ such transitions with labels $12, 13, ..., 1n$. Furthermore, following the rule, for each such transition $1k'$, the register mapping gets updated to $\rho' = \rho[1 \to k']$. Overall, due to $FINP$ there are n new state pairs $((P2, \rho'), (Q2k', \rho'^{-1}))$ that need to be recursively checked.

Consider some pair $((P2, \rho'), (Q2k', \rho'^{-1}))$. Again using $FINP$ rule, $n-1$ new mappings ρ'' are generated, where $\rho'' = \rho[2 \to k'']$ for all $k'' \in [n] \setminus \{k'\}$. Note that unlike for $Q1$, with $Q2k'$ the branching is limited.

In conclusion, starting with $5+3n$ nodes and approximately $n+2*n^2$ transitions, the resulting state space explored by $BISIM$ has approximately $n*(n-1)*(n-1)$ vertices. The result is that this example produces all the possible partial bijections. Naturally, this can be extended from size 3 to size n. $\triangle$

However, the scenario represented presented in the Example 3.14 does not appear to be possible with $\times\pi$-calculus. It is possible to produce a pattern similar to the example. However, in all of the attempts by the author, the growth due to $FINP$ rules was insignificant compared to the overall number of states and transitions in the initial LTS. In particular, the left system in the example only had locally fresh transitions, while with $\times\pi$-calculus there must be additional known input transitions for all the names stored in the registers. Additionally, by the transition rules, a fresh input is inserted in the smallest available register. Thus, to achieve the increasing pattern $1^\bullet, 2^\bullet, 3^\bullet, ...$, it is necessary to keep the fresh inputs in the registers throughout the execution. This implies additional known input transitions. Hence, the growth shown in the example is unlikely to occur in isolation.

3.5 Additional details

In this chapter, a local algorithm for deciding n-bisimulation was presented. The key downside of the approach taken is the determined worst-case time complexity. First, the lower bound of the complexity is limited because the algorithm is local. With the classical problem of bisimulation, global partition algorithms have significantly better worst-case time complexity than on-the-fly algorithms. Second, the approach suffers from potential combinatorial explosion. The reason for both aspects is the need for maintaining partial bijections.

As long as partial bijections are represented explicitly in the algorithm, and as long as each state has an attached partial bijection, the worst-case time complexity should account for the possibility of needing to explore the whole combinatorially large state space. This also applies to any potential global algorithm working with the presented state space.

With regards to adapting a partition algorithm, it would be undesirable to need to generate the whole state space as then a local algorithm would be a better approach. The author could not find a way of efficiently generating the necessary state space that would be better than the on-the-fly approach in the presented local algorithm. In particular, the issue stems from needing to deal with locally and globally fresh transitions. These transitions are covered by $INP2$, $FINP$, and $FOUT$ rules. The complexity comes from these rules updating the register mapping, where the update depends on the label of the matching transition.

On the positive side, while the algorithm presented determines strong n-bisimilarity, it can also be used for checking observational equivalence using the strategy described in the Background chapter in Section 2.5.1 by running it on a weakly-transformed LTS. However, applying it directly to the weakly-transformed LTS is not optimal. A more efficient and practical approach to weak bisimulation is discussed later in 4.2.5.

Chapter 4

Implementation

This chapter describes the tool implemented as a part of this dissertation, its supported functionality, and the implementation details.

4.1 Tool overview

The output of this dissertation is a command-line program named `pisim22`. The key functionality implemented in the program is the ability to check for equivalence of π-calculus models.

The tool takes in as input two π-calculus models described using ASCII text files. Since the tool makes use of `pifra` for its front-end processing, the input format and hence the syntax for the π-calculus models is that of `pifra`. The tool's output is a Boolean value indicating whether the passed models are equivalent or not as per early bisimulation.

The tool has support for the following additional functionalities:

- Observational equivalence – the tool can check for observational equivalence by checking for weak bisimulation.
- State reduction – the tool supports an additional state reduction strategy through garbage collection as defined in `pifra`.
- Control over execution parameters – the tool allows the user to override the default execution parameters such as the limit to the maximum number of states in a single LTS or the number of registers in n-bisimulation.
- Graphical representation of bisimulation relation – the tool can output a GraphViz DOT graph showing the merged LTS with the bisimilar states matched via additional edges. This feature can only practically be used for very small LTSs, but it can be useful in an educational setting when learning the basics of bisimulation.

- Execution statistics – various statistics, timing information and counters are tracked by the tool and available for users to inspect.

Example 4.1 Using the tool. For an example of usage of the tool, consider two simple π-calculus models. The first is $i(x).\overline{o}\langle x\rangle.0$, and the seconds one is $\nu c.(i(x).\overline{c}\langle x\rangle.0 \mid c(x).\overline{o}\langle x\rangle.0)$. Two models are observationally equivalent. To check these two models, they need to be expressed in the `pifra` syntax. The input files are given below. The input for the first model is given in Listing 4.1:

```
1 i(x).o'<x>.0
```

Listing 4.1: File jev-example.1.pi.

The file for the second model is given in Listing 4.2:

```
1 $c.(i(x).c'<x>.0 | c(x).o'<x>.0)
```

Listing 4.2: File jev-example.2.pi.

The models can be checked for either strong or observational equivalence. The command for and results of checking for strong equivalence are given in Listing 4.3:

```
1 $> ./pisim22.exe -lts1 jev-example.1.pi -lts2 jev-example.2.pi
2 ^^^ Systems are NOT bisimilar for rho map[1:1 2:2], N=2.
```

Listing 4.3: Strong equivalence check of jev-example models.

Two models are expectedly not strongly equivalent because the second models contains an internal actions. Checking for observational equivalence is given in Listing 4.4:

```
1 $> ./pisim22.exe -lts1 jev-example.1.pi -lts2 jev-example.2.pi -w
2 *** Systems are BISIMILAR for rho map[1:1 2:2], N=2.
```

Listing 4.4: Observational equivalence check of jev-example models.

The only difference is the `-w` flag triggering weak bisimulation. Two models are expectedly observationally equivalent. To better understand the result, the tool can output the bisimulation relation built – the vertices in graph G. For small systems, this bisimulation relation can be visualised. For the models in this example, the visualisation of weak bisimulation is given in Figure 4.1. The first model is on the right side, while the second model is on the left side of the figure. Dotted lines between the states represent the bisimulation relation. Note that the output graph contains the normalised names.

△

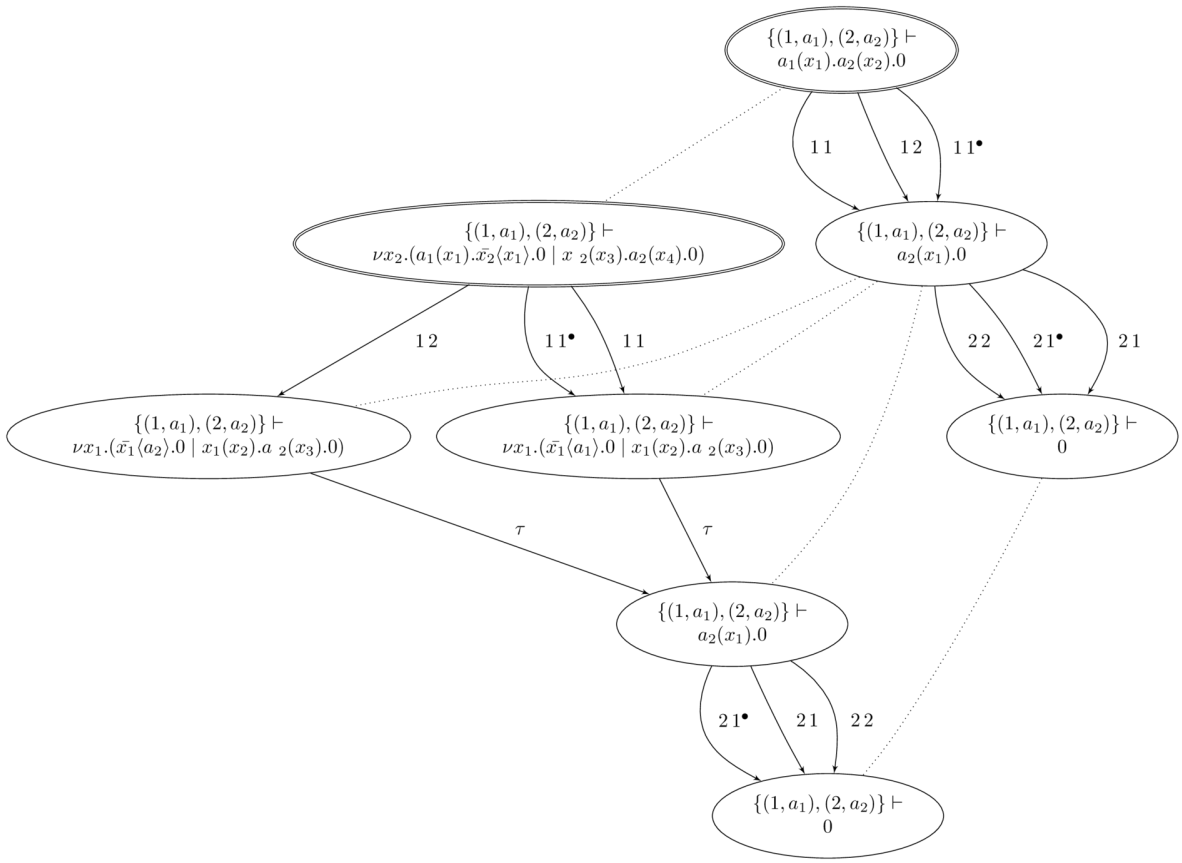

Figure 4.1: Bisimulation graph for jev-example systems. Dotted lines represent the bisimulation relation.

4.2 Implementation details

The tool is fully implemented in the Go programming language. The choice of a programming language was pragmatic and was based on the fact that `pifra` was implemented in Go and is available as a Go library. None of Go's unique features or qualities is used in the implementation. Thus, any other general-purpose programming language could have been suitable. In particular, implementation could have benefited from a language with a more extensive standard library for the commonly used data structures.

4.2.1 Implementation overview

As mentioned in Section 3.1, the approach to equivalence checking of π-calculus consists of multiple layers.

The high level tool `pisim22`, consists of three main components: Leung's tool `pifra`, a component for generating a weakly-transformed LTS (§2.5.1), and a component implementing the bisimulation checking that was presented in the previous chapter (§3.2).

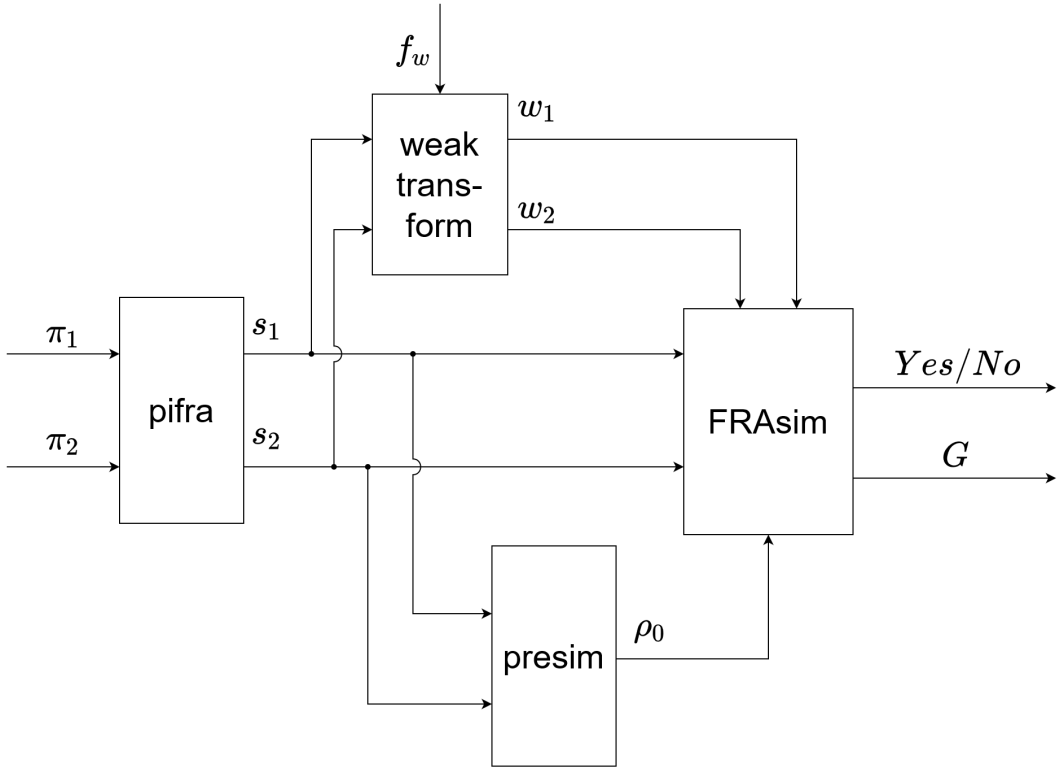

Figure 4.2: High-level architecture of the implementation.

The overall architecture of the tool can be represented as a pipeline performing a sequence of operations. An architecture diagram can be seen in Figure 4.2. The implementation can be split into four main distinct parts, as can be seen in the diagram. The input to `pisim22` are two π-calculus models denoted as π_1 and π_2, each represented as a file with an ASCII string following the syntax required by `pifra`. These two models are individually run through `pifra` to produce $\times\pi$-calculus LTSs s_1, s_2, which can also be considered to be FRAs. If a flag f_w is set, then the produced LTSs are individually run through `weak transorm` that produced weakly-transformed LTSs w_1 and w_2. Weakly-transformed LTSs are needed when checking for weak n-bisimulation. If only strong n-bisimulation is needed, then the f_w is set to false, and `weak transorm` acts as a no-op. As mentioned in Chapter 3, before checking for n-bisimulation, one needs to construct an initial mapping between the starting registers of s_1 and s_2. To do that the LTSs s_1, s_2 are passed to `presim`, which determines the value of n for n-bisimulation and constructs the initial mapping ρ_0. With all the preprocessing done, the bisimulation algorithm can be run, and this is represented by the `FRAsim` component. The output is then a Boolean value indicating whether two models are bisimilar or not, and the bisimulation graph G.

Remark. Note that not of the functionality is demonstrated in the diagram. Only the key process of bisimulation checking is shown.

Below, relevant details concerning each of the components are given.

4.2.2 Pifra

As mentioned previously (§2.8.1), `pifra` has one big issue that does not allow it to be directly plugged into the presented architecture – the LTS produced by `pifra` does not retain enough information about the channel names. Due to normalisation, only the relative alphabetical order of the free channel names is retained, while the names themselves are forgotten. This is an issue as the correctness of n-bisimulation relies on the correct mapping ρ between the registers. However, if normalised names are used, then this leads to a large number of false positives. For an example, see Example 2.28.

Other normalisations performed by `pifra` do not affect the correctness of the mapping as they only normalise the names, but they do not change in any unexpected way the locations of specific names between transitions.

To fix this issue, `pifra` was slightly modified. The fix relies on the observation that only names stored in registers of the starting states are required to build the initial mapping ρ_0. Hence, it is enough to create a mapping between the original names and normalised names. This can be created when the names are first normalised. To make use of this mapping, it is added to the returned `Lts` object (see Listing 4.5).

Additionally, a critical bug was found and fixed during the development and testing. The bug was due to a missing counter increment, and it led to some systems not being identified as equivalent. It occurred when handling the $CLOSE$ rule, leading to a possible register override when handling the RES rule.

4.2.3 Weak transform

This component deals with constructing a weakly-transformed LTS (§2.5, §2.5.1). The computation is performed in two steps.

First, a transitive closure on the τ-transitions of the original LTS is computed to obtain $\Rightarrow$. Practically, the closure is found by doing a depth-first search (DFS) from each node to find the states reachable by just consecutive τ-transitions. For a graph (V, E), the complexity of this approach is in $O(V^2 + V * E)$. An alternative also explored was the Floyd-Warshall algorithm, which, however, performs poorly on sparse graphs due to its complexity being in $O(V^3)$. An algorithm can be chosen by the user depending on the specifics of their models.

The second step is to obtain $\stackrel{a}{\Rightarrow}$. Since $\stackrel{a}{\Rightarrow}$ strictly contains all the $\stackrel{a}{\rightarrow}$, the algorithm takes the original LTS as a starting point. The remainder of the transitions are obtained by composing $\stackrel{a}{\rightarrow}$ with $\Rightarrow$. That is, for every transition $u \stackrel{a}{\rightarrow} v$, if there exists u' and v'

such that $u' \Rightarrow u \xrightarrow{a} v \Rightarrow v'$, then $u' \overset{a}{\Rightarrow} v'$ is added to the weakly-transformed LTS. Note that it is possible that $u = u'$ and/or $v = v'$. A direct implementation of this algorithm has a complexity in $O(E * V^2)$.

The overall complexity of `weak transform` is then in $O(E * V^2)$ when used with the DFS closure algorithm. Thus, for an input LTS $\langle S, A, \rightarrow \rangle$, `weak transform` stage takes time in $O(|\rightarrow| * |S|^2)$, which in the worst case is in $O(4n * |S|^4)$.

Overall, the resulting weakly-transformed LTS has an increased number of transitions. The specific increase factor depends on the amount of internal activity in a specific model. In 4.2.5 it is explained what changes are needed to make the weak bisimulation checking more practical.

4.2.4 Presim

The `presim` component determines the initial mapping ρ_0. Some details were already given in Section 3.2.2. From an implementation side, this component is rather trivial.

First, the value n is picked as the maximum size of the register sets among the two passed FRAs. Practically, this might be excessive as an FRA can have more registers than it can use. Therefore, the value n is chosen to be the maximum number of names stored in registers across all the states of two FRAs. This is relevant as, for example, `pifra` assumes an effectively infinite number of registers. Nevertheless, a user is given an option to override the value of n.

Second, ρ_0 can be constructed by simply matching the same names across the registers of the two starting states.

4.2.5 FRAsim

Within the `FRAsim` component, the bisimulation algorithm that was presented and explained in Chapter 3 is implemented. The algorithm's logic was already explained, and the implementation directly follows it. The aspects considered here are the data structures used and other optimisations for efficient implementation.

Data structures

The primary data structure in the pseudo-code specification of the algorithm is a set. Many sets are being maintained: a set of not-related states $\overline{R}$, graph G which effectively consists of a set of vertices $V(G)$ and set of edges $E(G)$, a set of transitions from a specific state in one of the LTSs, a set A with information about the states needing re-evaluation, and the sets with pointers. A naive approach to set membership would be to traverse

the whole set sequentially, filtering away the entries not matching the condition. This, however, is very costly, especially since conditions often only match a few entries of a big set. Thus, an alternative approach was taken.

The LTS produced by `pifra` simply maintains the graph as a set of states and a list of transitions. Each unique configuration is associated with an integer ID. The uniqueness of the configuration is practically assessed by having a string key that fully identifies the configuration. The data structures are given in Listings 4.5, 4.6. Note that `FreeNamesMap` in `Lts` was added by the author.

```
type Lts struct {
  States       map[int]Configuration
  Transitions  []Transition
  FreeNamesMap map[string]string
}

type Transition struct {
  Source      int
  Destination int
  Label       Label
}

type Label struct {
  Symbol  Symbol
  Symbol2 Symbol
}
```

Listing 4.5: The lts data structures from `pifra` 1/2. [18]

```
type Configuration struct {
  Process   Element
  Registers Registers
  Label     Label
}

type Registers struct {
  Size      int
  Registers map[int]string
}

type Symbol struct {
  Type  SymbolType
  Value int
}
```

Listing 4.6: Original data structures from `pifra` 2/2. [18]

This way of representing a graph requires iterating over all the transitions at all times. As explained, this is unacceptable for the bisimulation algorithm. Instead, an extended variant of adjacency list representation is used for LTSs. It is presented in Listing 4.7.

```
type AdvAdj = map[int]map[LabelsKey][]pifra.Transition

type LabelsKey struct {
  SymbolType1 pifra.SymbolType
  SymbolType2 pifra.SymbolType
}
```

Listing 4.7: Advanced adjacency list data structure.

The main feature of this data structure is that the list of transitions is first partitioned by the originating vertex as in the classical adjacency list, and then it is also partitioned by the type of the label, which is a combination of the following symbols: $\{i, \bar{i}, i^{\bullet}, i^{\circledast}, \tau\}$. The benefit is that the search space for building the *derivs* set in $MATCH\_LEFT$ (§3.2) is reduced.

Remark. Note that this is still not optimal as, for example, in rule INP1, the partitioning can be even more granular based on the exact label and not just label type.

The above covers the representation of the original and weakly-transformed LTSs. Other sets deal with ν-configurations, which are represented by a `FRAConfiguration` data structure that extends `Configuration` from `pifra`. It can be seen in Listing 4.8.

```
1 type FRAConfiguration struct {
2   Process   pifra.Element
3   Registers pifra.Registers
4   Rho       map[int]int
5   N         int
6 }
```

Listing 4.8: Data structure for ν-configurations.

Similarly to how `pifra` handles configurations, each ν-configuration has an associated unique textual representation. It should be noted that since the LTSs for two models are not truly merged, two states with the same textual representation can appear. Hence, to create a unique key, an additional piece of information about the model from which the state originated needs to be added. A key for a pair of ν-configurations is a simple concatenation of individual keys. These textual keys are used as identifiers, especially when a hashable identifier is required. Sets $\overline{R}$ and $V(G)$ then simply have a type `map[string]bool` representing a simple set.

More care is required for the $E(G)$ set, which is used for more fine-grained membership checks. Similarly to transitions in an LTS, the solution reduces the search space by partitioning. The access pattern to $E(G)$ requires filtering by the source vertex when filling the set A, and it requires filtering by both the source and destination vertex when removing unnecessary edges after removing a vertex from the graph. The solution is to use a data structure presented in Listing 4.9.

```
1 type gGraph struct {
2   States map[string]gVertex
3   TransitionsSrcMap map[string]map[string]*gTransition
4   TransitionsDstMap map[string]map[string]*gTransition
5 }
```

Listing 4.9: Data structure for the graph.

Two nested hash tables are kept for transitions in this graph data structure. One is partitioned by the source vertex, and the other is partitioned by the destination vertex. The second layer is a map from transition IDs to pointers to the identified transitions. Since pointers are used, the increase in used memory due to duplicate transitions is limited. A map is used to support fast random element insertion and deletion.

The rest of the data structures follow directly from the pseudo-code. Wherever a hashable key is required, a corresponding identifying textual representation is used.

Speeding-up weak bisimulation

Classical weak simulation was presented in Section 2.5 in Definition 2.13. Implementing weak bisimulation using this definition is trivial as it simply amounts to running the strong n-bisimulation algorithm on a weakly-transformed LTS without any changes to the algorithm itself. However, this is inefficient as doing a weak transformation can significantly increase the number of edges in an LTS.

A more efficient way of doing weak bisimulation is to use an alternative definition of weak simulation that was given in Definition 2.14. The critical difference is that in this definition, strong transitions are matched with weak transitions, where strong transitions are those in the original LTS, and weak transitions are those in weakly-transformed LTS. Meanwhile, weak transitions are being matched with weak transitions in the inefficient definition, increasing the search space.

To implement the latter more efficient definition, the program needs to store in memory both the original LTS and the weakly-transformed LTS. In practice, this optimisation proved to be very important for speeding up weak bisimulation.

Chapter 5

Evaluation

This chapter presents the details of how the implementation was evaluated. In the rest of the chapter, some details about the test cases used are given, results are presented and discussed, comparison with some other tools is considered.

5.1 Experiments

More than 30 examples were used to test `pisim22`, including all: strong and weak equivalences and inequivalences. The primary purpose was to test the correctness of the implementation. Secondary, the practicality and scalability of the tool were evaluated using some of the test cases as benchmarks.

When testing the implementation, some of the examples were adapted from another process calculus – a calculus of communicating systems (CCS).[22] It is also due to Milner, and it is a predecessor of π-calculus. It is a less powerful calculus, which unlike π-calculus does not support name passing. Hence, all the examples from CCS can also be expressed in π-calculus and are helpful for testing of basics of bisimulation.

Other examples come mainly from literature and represent more interesting systems that can be modelled with π-calculus. These examples also demonstrate practical applications of π-calculus and equivalence checking.

The set of test cases consisted of the following examples:

- Small scale test cases from the literature. These are testing the basics of the definitions of strong and weak bisimulation. They are not testing for π-calculus features but rather for general CCS. Examples were mainly taken from books by Milner [25] and Sangiorgi [35], [36]. These test cases normally include `milner` or `sangiorgi` in their names.
- Small scale test cases testing for all the aspects of n-bisimulation. Most of these

test cases were created by the author to ensure full coverage of the implementation logic. These include a prefix `jev`.

- Milner's job shop [25, Section 7.2] – represents a more complex example of observational equivalence in CCS.
- Milner's scheduler [25, Section 7.3] – a well known example of observational equivalence in CCS. This problem is of particular interest as it is parametric in the number of agents. Moreover, its models are defined via simple inductive formulae, making them suitable for programmatic generation. Thus, it is perfectly suitable as a benchmark. It has been used for benchmarking the performance of the Aldébaran tool.[11][10]
- Simple buffer – a first-in-first-out buffer, effectively a queue that receives values on one channel and outputs the names on the second channel. In this test case, an implementation using a composition of n buffers of size one is compared with the specification of the n-buffer. This example is parametric as the size of the buffer can be varied. Unlike Milner's scheduler, buffer makes more extensive use of π-calculus features as its inputs and temporarily stores names from an infinite alphabet.
- Alternating bit protocol – it is a simple protocol whose specification is that of a simple buffer. Its implementation, however, uses retransmission to achieve reliable unidirectional transmission. The explanation and specification of the alternating-bit protocol can be found in [23]. The specific implementation used is taken from the example set of The Mobility Workbench. The test cases have a prefix `cleav-abp`.
- The GSM handover protocol – the well-known example of a handover procedure in GSM Public Land Mobile Network. The specifications are from [31]. Some variants of this example have been used for testing in both MWB[38] and HAL[13] tools. Here, three models were used – one for the simple specification of the protocol. And two for the implementation with and without the error handling.
- Boolean logic – an example of representing truth values in π-calculus is given in [25, Section 10.3], the implementation is adapted from the MWB examples set. Here only a simple test case using a negation operator is used. More complex examples using boolean logic can be constructed.

Remark. As mentioned previously, π-calculus is computationally universal, and it can relatively easily be used to represent more interesting examples. However, as is demonstrated by the results, the tool already struggles with moderately large models. Hence, it cannot practically work with many of the models. Only test cases for which the tool finished in a reasonable time are included here.

Name	$\|S_1\|$	$\| \rightarrow_1 \|$	$\| \rightarrow'_1 \|$	$\|S_2\|$	$\| \rightarrow_2 \|$	$\| \rightarrow'_2 \|$	n	$t(pifra)$	$t(bisim)$	$t(total)$
Job Shop	65	437	502	1509	5222	29527	6	6s	0.7s	6.7s
ABP-JP	7	12	19	1152	2477	116330	4	10s	1s	11s
GSM 1	3910	7127	118388	364	796	1520	5	1m54s	3s	1m57s
GSM 2	3939	7219	143531	364	796	1520	5	2m49s	3s	2m52s
GSM 3	3910	7127	118388	3939	7219	143531	5	4m40s	37s	5m17s
Sched(5)	695	21105	36255	321	11051	11372	11	13.2s	3.5s	16.7s
Sched(6)	1679	70862	117806	769	36301	37070	13	1m24s	13s	1m37s
Sched(7)	3935	220403	357405	1793	111119	112912	15	8m42s	52s	9m34s
Sched(8)	9023	647780	1029920	4097	322577	326674	17	53m30s	3m35s	57m5s
Buffer(3)	490	1028	3960	296	656	952	5	0.27s	0.15s	0.42s
Buffer(4)	5296	12262	67830	2829	6203	9032	6	7.6s	2.8s	10.4s
Buffer(5)	69261	167716	1324919	34637	74892	109529	7	4m4s	5m20s	9m24s

Table 5.1: Running time details for more complex models requiring weak bisimulation. Garbage collection is **disabled**.

All the test cases used can be found together with the source code as described in the Source Code chapter.

5.1.1 Performance

Performance was evaluated on all the test cases mentioned previously. In particular, Milner's scheduler and the buffer examples were used as they allow to control the size of the problem parametrically.

To measure the performance, a machine with an Intel Core i5-1135G7 processor and 16GB of RAM running Windows 10E 21H2 was used. The key metric recorded is the time to run the program for each test case. In particular, since time was found to be the limiting factor. Since the program is structured as a sequence of operations, the times per each stage are also recorded. Moreover, for judging the size of the problems, the sizes of the LTSs and other relevant parameters are given.

The results for some of the more complex test cases are presented in Table 5.1. The tool was set to check for weak bisimulation, and garbage collection was turned off. The columns are as follows: the name of the test case, the total number of states in the first LTS as produced by `pifra`, the total number of transitions in the first LTS as produced

Name	$\|S_1\|$	$\|\rightarrow_1\|$	$\|\rightarrow'_1\|$	$\|S_2\|$	$\|\rightarrow_2\|$	$\|\rightarrow'_2\|$	n	$t(pifra)$	$t(bisim)$	$t(total)$
Job Shop	29	168	197	700	2358	13328	6	3s	0.3s	3.3s
ABP-JP	4	6	10	537	1113	48679	4	5s	0.4s	5.4s
GSM 1	1586	2722	39007	163	316	626	5	46s	1s	47s
GSM 2	1597	2751	45353	163	316	626	5	1m11s	1s	1m12s
GSM 3	1586	2722	39007	364	2751	45353	5	1m52s	13s	2m5s
Sched(5)	248	10104	17371	161	5291	5452	10	21s	1.7s	22.7s
Sched(6)	804	34135	56776	385	17485	17870	12	3m22s	6s	3m28s
Sched(7)	1968	106674	173037	897	53775	54672	14	28m	23s	28m23s
Buffer(3)	215	400	1522	133	255	388	5	0.13s	0.06s	0.19s
Buffer(4)	2344	4794	25674	1271	2459	3730	6	3.6s	1s	4.6s
Buffer(5)	31452	67641	510742	15529	30161	45690	7	1m51s	45s	2m36s
Buffer(6)	>100K	>200K	-	>100K	>200K	-	-	-	-	-

Table 5.2: Running time details for more complex models requiring weak bisimulation. Garbage collection is **enabled**.

by `pifra`, number of transitions in the weakly-transformed LTS, the total number of states/transitions/weak-transitions for the second LTS, the size of registers set n, time taken for `pifra` front-end to execute, time taken for the bisimulation checker part to execute (also includes weak transformation), total time taken to run the command.

The tests cases used are as introduced above. In particular, *Job Shop* is for Milner's job shop example. *ABP-JP* is for the model that matches the specification of the alternating bit protocol. *GSM* is for the GSM handover protocols. Specifically, *GSM 1* refers to checking implementation without error correction against the specification, *GSM 2* is for implementation with error correction against the specification, and *GSM 3* is for checking both implementations against each other. *Sched(N)* denotes the Milner's scheduler examples where N is the number of agents. *Buffer(N)* denotes a test case for the buffer of size N.

The previous table listed results with garbage collection disabled. For results with garbage collection enabled see Table 5.2. Columns have the same meaning. Note that an entry for *Sched*(8) is missing as it took more than two hours to run.

5.1.2 Performance analysis

Consider the results in Table 5.1. All the problems there required the use of weak bisimulation. The sizes of the transition sets reveal that weak transformation can cause a large increase in the number of transitions. The largest increase happens for the *ABP-JP* test case, where the number of transitions grows by more than 46 times. However, the increase is variable and depends on the amount of internal activity. In particular, the increase in the scheduler test cases is less than twofold. This highlights the importance of using a more efficient weak bisimulation algorithm (§4.2.3) to reduce the time of checking for weak bisimulation.

To analyse the performance in more detail, a closer look at the results for the parametric test case of Milner's scheduler system is taken. A plot of total execution time against the problem complexity parameter N, corresponding to the number of agents, is given in Figure 5.1. This plot also contains a separate curve for the time taken to do just the bisimulation check, i.e. total time minus time taken by `pifra`. The plot reveals rapid growth in the time needed with an increase in the number of agents. The curves displayed are polynomials of degree 7 for total time and degree 5 for bisimulation time.

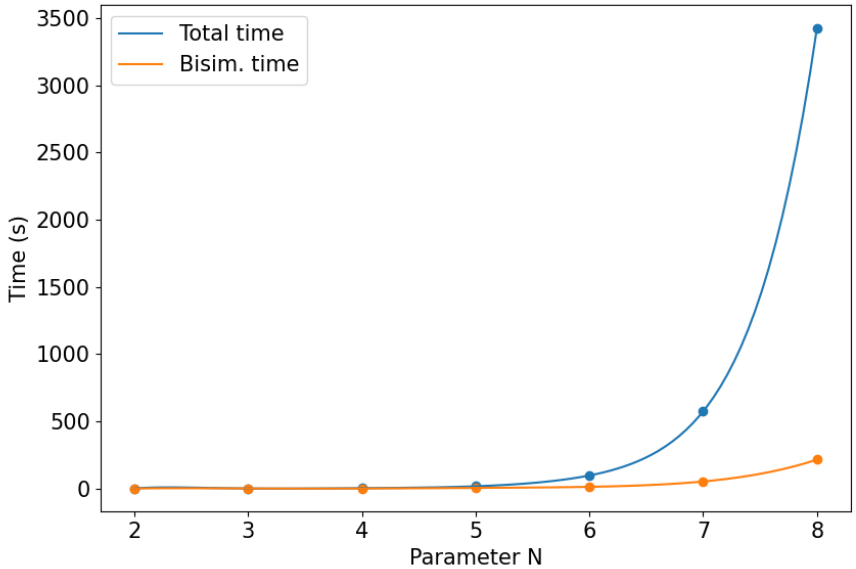

Figure 5.1: Scheduler performance: time vs number of agents (N).

However, it is more interesting to consider the time taken with respect to the number of states in a strong LTS. In particular, a product of the states $|S_1| * |S_2|$ is used. Consider such a plot in Figure 5.2. While the previous plot showed a high-degree-polynomial growth, this plot reveals a more moderate linear growth. In particular, the plot clearly

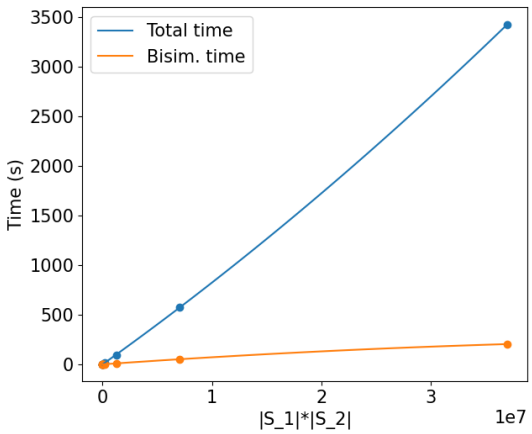

Figure 5.2: Scheduler performance: time vs $|S_1| * |S_2|$.

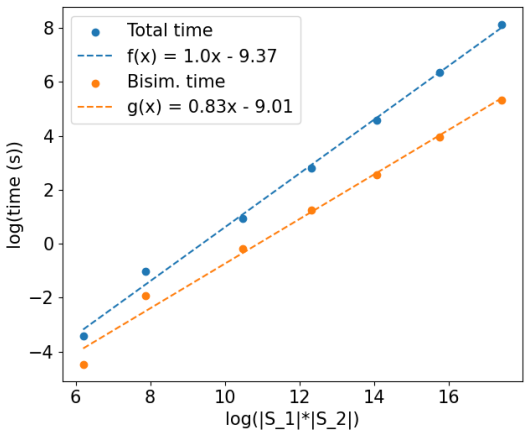

Figure 5.3: Scheduler performance: log-log plot of time vs $|S_1| * |S_2|$.

demonstrates that `pifra` amounts for the majority of time necessary for the program to run. To better understand the growth, consider a log-log plot for the same data in Figure 5.3. The curves again look like straight lines. A simple line fitting is used, and the equations for the fitted lines are presented in the figure. These reveal that the growth is approximately s^1 for the total time and $s^{0.83}$ for the bisimulation time, where s is the product of states.

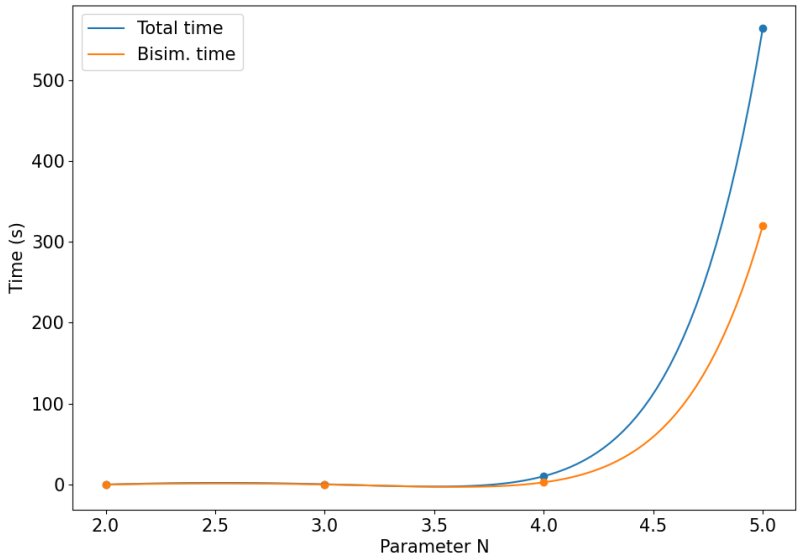

Figure 5.4: Buffer performance: time vs buffer size (N).

For slightly different results, consider the buffer example. A plot of time against the buffer size can be seen in Figure 5.4. The meaning of the plot is the same as for the

scheduler example. The curves displayed are ninth-degree polynomials. From the plot, it can be observed that the time for bisimulation grows almost identically to the total time for buffer. That is, for buffer `pifra` is not a bottleneck, unlike for scheduler. Otherwise, buffer test cases also show a very rapid growth, which is greater than for scheduler.

Similarly, consider the change in time versus the change in the size of a strong LTS measured as a product of the size of state sets: $|S_1| * |S_2|$. A plot of time against the size of a strong LTS can be seen in Figure 5.5. It reveals a close to linear growth in both total and bisimulation time with respect to the product $|S_1| * |S_2|$. This is confirmed by a log-log plot which can be seen in Figure 5.5. The log-log plot reveals that growth can be approximated by $s^{0.7}$, where $s = |S_1| * |S_2|$.

As mentioned, buffer examples use the name passing feature π-calculus, unlike the scheduler examples. Specifically, the buffer example represents the limitations of using FRA. It is reflected in a large number of states and their rapid growth. Notice that the LTS for the schedulers can be described as being dense graphs, while for buffers, the graphs are sparse.

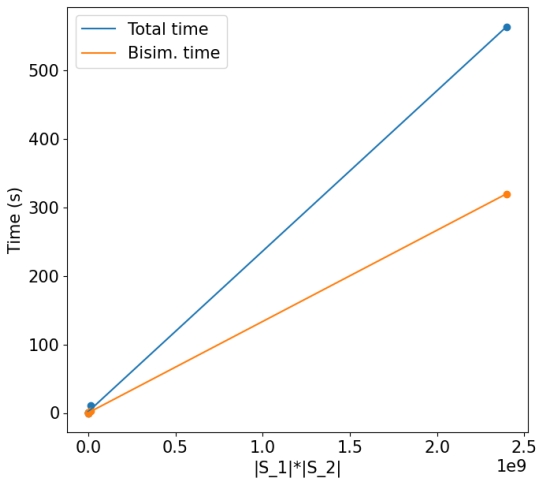

Figure 5.5: Buffer performance: time vs $|S_1| * |S_2|$.

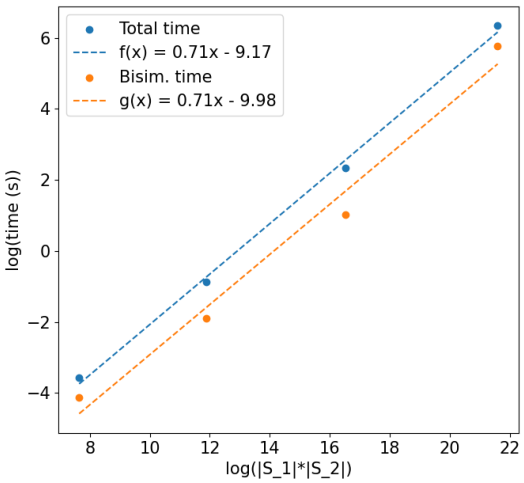

Figure 5.6: Buffer performance: log-log plot of time vs $|S_1| * |S_2|$.

The results from the scheduler and buffer test cases suggest that the practical complexity of the tool is linear with respect to the product of sizes of state sets. Consider a log-log plot of times for a larger set of test cases against $|S_1| * |S_2|$ in Figure 5.7. It confirms the previous observations. In particular, in all the realistic test cases examined, no exponential or factorial growth with respect to n - the size of the register set - was observed. This suggests that the worst-case complexity as calculated previously (§3.4) might not often occur in practice.

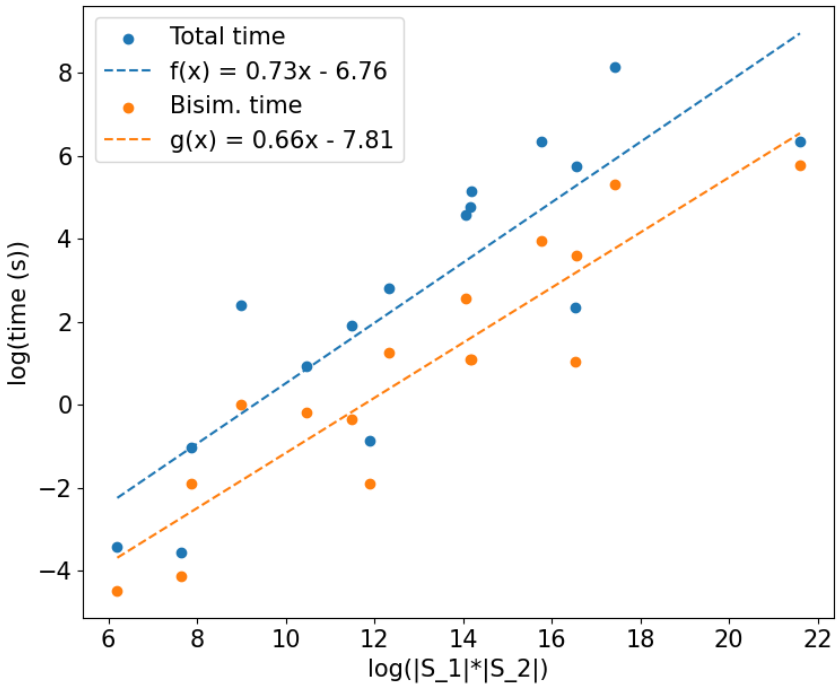

Figure 5.7: All performance: log-log plot of time vs $|S_1| * |S_2|$.

Since the size of the strong LTS affects the execution time, it is natural to investigate the effects of enabling garbage collection. The main idea behind garbage collection is to reduce state space, which in theory should also decrease the time necessary for checking the equivalence. The change introduced by garbage collection can be seen in Figure 5.8. The bar chart shows a percentage change in total time taken with respect to the total time with garbage collection disabled. The chart reveals that the change in total time is not always negative. Specifically, enabling garbage collection increased the time needed for all the instances of Milner's scheduler problem. In fact, the slow-down grows with the problem size for the scheduler. On the other hand, there was about a twofold reduction in time for all the other examples.

The details in the results tables show that the sizes of strong LTSs have reduced for all the examples as expected. And this reduction also meant that $t(bisim)$ decreased as well. However, having garbage collection enabled leads to an increase in $t(pisim)$ for all the instances of the scheduler problem. It appears that the structure of the scheduler model triggers the worst-case performance in `pifra`. This could be because the scheduler specification includes a large number of summation operators.

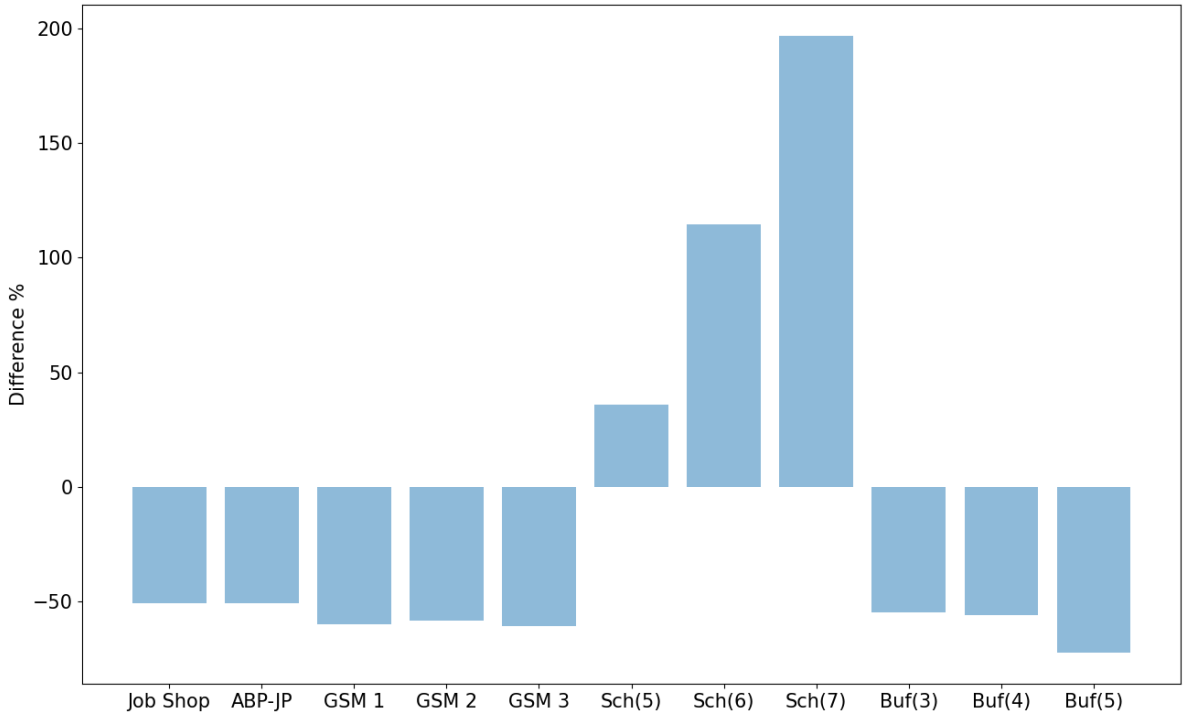

Figure 5.8: Effect of garbage collection on performance. Bar chart shows percentage difference in total time taken resulting from having garbage collection enabled.

5.2 Comparisons with other tools

One of the main goals of this dissertation was to implement a way of checking early bisimulation. This is in contrast to other more restrictive equivalence relations commonly used in other tools. For example, open bisimulation in MWB or the bisimulation relation in `pisim`. The tool implemented does indeed check for early bisimulation. Some examples of how `pisim22` compares with competitors are presented in the rest of this section.

5.2.1 Comparison with MWB

One of the best well-known tools that can also check for equivalences between π-calculus models is the Mobility Workbench (MWB). As mentioned (§2.7), it uses a more restrictive open bisimulation relation.

An example of two processes that are bisimilar under early bisimulation but not under late and hence also not under open bisimulation is given by the two files in Listings 5.1 and 5.1. The example is originally due to Sangiorgi. [35] The key aspect of this example is the presence of a match expression, the result of which depends on the received name.

```
1 PP = a(x).b'<x>.0 + a(x).0
2 PP
```

Listing 5.1: File jev-sangiorgi-open-bisim.1.pi.

```
1 QQ = a(x).b'<x>.0 + a(x).0 + a(x).[x=true]b'<x>.0
2 QQ
```

Listing 5.2: File jev-sangiorgi-open-bisim.2.pi.

When checking these two models for equivalence using `pisim22`, the result is that they are determined to be equivalent. The command for running this check and the output can be seen in Listing 5.3.

```
1 $> ./pisim22.exe -lts1 jev-sangiorgi-open-bisim.1.pi -lts2 jev-sangiorgi
     -open-bisim.2.pi
2 *** Systems are BISIMILAR for rho map[1:1 2:2], N=3.
```

Listing 5.3: Open bisimulation example in pisim22.

On the other hand, when evaluating the same models using MWB, the produced result is that models are not equivalent. The definitions in the MWB syntax, as well as output and the command for equivalence checking, can be seen in Listing 5.4. For this test, MWB99 with a version dated 2006-08-19 was used.[1] To run it, New Jersey Standard ML compiler version 110.59 was used.

```
1 MWB> agent PP(a,b) = a(x).'b<x>.0 + a(x).0
2 MWB> agent QQ(a,b,true) = a(x).'b<x>.0 + a(x).0 + a(x).[x=true]'b<x>.0
3
4 MWB> eqd (true) PP(a,b) QQ(a,b,true)
5 The two agents are NOT equal.
```

Listing 5.4: Example of limits of open bisimulation in MWB.

Nevertheless, MWB is a tool with a larger number of features than `pisim22`. In particular, MWB can deal with certain models with non-finite-control as its on-the-fly checking strategy can detect inequalities even in some infinite models. The current implementation of `pisim22` is, however, limited to models for which `pifra` can generate a finite LTS. Moreover, MWB appears to be faster than `pisim22`. For example, it takes MWB less than 4 seconds to calculate *GSM 2* test case, while it takes over a minute for `pisim22`. However, a more rigorous investigation is needed to make a definite conclusion.

[1]The tool and instructions were obtained from `https://www.it.uu.se/research/group/concurrency/mwb`.

5.2.2 Comparison with the original pisim

In Section 2.8.2, two examples of how the equivalence checking logic in `pisim` is not optimal were given. First, there are false negatives as the classical bisimulation used requires register contents to match exactly. This was tackled in `pisim22` by using n-bisimulation. Second, there are false positives due to using normalised names outputted by `pifra`. This was tackled by updating `pifra` to retain the necessary information about the names. Both of these examples were tested with `pisim22`, and the results matched the expected outputs. Hence, `pisim22` was determined to be a significant improvement over the original `pisim`.

5.2.3 Comparison with PiET

The key difference worth highlighting is how the tools deal with observational equivalence. In `pisim22` classical notion of weak bisimulation is implemented. Meanwhile, it is not completely clear what notion of weak equivalence is used in `PiET`. In particular, consider the following example due to Sangiorgi [35, example 4.2.6]. The processes $P = \overline{a}\langle a\rangle.0$ and $Q = \tau.0 + \tau.\overline{a}\langle a\rangle.0$ are not observationally equivalent using the notion of weak bisimulation. This is because Q can terminate without producing any visible actions by transitioning as $Q \Rightarrow 0$. However, in `PiET`, those two processes are determined to be "*Weak Early Equivalent*". An example of inputs and outputs to `PiET` can be seen in Listing 5.5. These results suggest that in `PiET` τ-transitions are simply ignored when testing for weak equivalence.

```
1 AGENT K2(a) := a<a>.0;
2 AGENT L2(a) := (_t.0 + _t.a<a>.0);
3 TEST K2(a) WITH L2(a);;
4 OUTPUT> YES. The two processes are Weak Early Equivalent.
```

Listing 5.5: Example of PiET not using classical weak bisimulation.

On the other hand, `pisim22` handles this example as expected. Note that since `pifra` does not support explicit τ-action, they are also not supported in `pisim22`. However, it is possible to generate τ-action through parallel processes and private channels like in Listing 5.6.

```
1 L2 = $p1.(p1(x).a'<a>.0 | p1'<p1>.0) + $p2.(p2(x).0 | p2'<p2>.0)
```

Listing 5.6: Constructing τ transitions in `pisim22`.

It should also be mentioned that weak bisimulation is not a congruence as it is not preserved under the summation operator. Thus, as expected, `pisim22` does not recognise $\tau.P + Q$ and $P + Q$ as being weakly equivalent, while it does for $\tau.P \approx P$.

5.3 Summary

In summary, the tool has succeeded in implementing an equivalence checker for π-calculus. It implements early bisimulation and thus compares favourably to tools implementing more refined relations such as open bisimulation in MWB. It also presents a significant improvement over the previous attempt in `pisim`. Additionally, it successfully implements the classical notion of observational equivalence through weak bisimulation.

The results of the performance testing show that real-world examples do not result in the worst-case complexity. However, the time taken to execute the test cases do not appear to be practical. For example, it cannot practically handle simple buffers of size six or larger. Overall, the translation performed by `pifra` appears to be a bottleneck. Nevertheless, more evaluation is needed to determine how the performance compares to other available alternatives and what are the key attributes characterising the performance of the approach presented here. Other than time and memory, it would be interesting also to consider the LTS size when translating to FRA compared to other approaches. For example, a relatively simple buffer example resulted in many states.

One of the performance evaluation limitations is that the time complexity is measured in terms of multiple variables. However, the parametric test cases had all the variables increasing simultaneously together with the changing parameter. And the rate of change was not the same for all the variables. To better understand the practical complexity, it would be desirable to investigate the change in time versus the change in a single variable only, keeping the rest of the variables fixed.

Chapter 6

Conclusion

6.1 Overview

The problem tackled in this dissertation was that of equivalence checking in π-calculus. Due to the infinite nature of π-calculus, this required symbolic reasoning, which was achieved through the use of an intermediate formalism – fresh-register automata (FRAs). The FRAs is an abstract model of computation over names involving fresh name creation. It is known that many π-calculus processes can be expressed as finite FRAs through the help of $\times\pi$-calculus. And it is known that bisimulation equivalence in π-calculus is related to n-bisimulation relation in $\times\pi$-calculus and FRA. Thus, this dissertation tackled the issue of practical equivalence checking in π-calculus through FRA.

In chapter 2, it was described how the work in this dissertation builds on the previous works of other students under the supervision of the same supervisor. In particular, it builds on the work by Leung, who, as a part of his dissertation work, created a tool `pifra` for translation from π-calculus to FRA through $\times\pi$-calculus. This tool became an essential part of the equivalence checker created as a part of this dissertation. However, Leung's tool required some fixes and adjustments to make it fully suitable for use with equivalence checking. The same chapter also described how another student attempted to tackle effectively the same problem as this dissertation but only achieved very limited success. The limitations of his approach were investigated, and counter-examples were provided to illustrate the problems. The solution produced in this dissertation significantly improves on this previous attempt. It eliminates false positives and reduces false negatives.

Since there was already a tool for translation from π-calculus to FRAs, the missing part was checking for n-bisimulation in the generated FRAs. While the definition of n-bisimulation is known, there was no description of an algorithm for practical checking for that equivalence. In chapter 3, a novel algorithm for checking n-bisimulation

was presented. This algorithm is built upon Celikkan's on-the-fly algorithm for classical bisimulation between LTSs. It extends it by making it work over a new type of states and incorporating the complex set of conditions for n-bisimulation. It was argued that the algorithm terminates and is indeed correct. A time complexity analysis was provided, which determined that due to the inclusion of partial bijection in the states, the algorithm has the potential to suffer from state explosion. However, the author was not able to find an actual example demonstrating the calculated worst-case complexity.

A more tangible contribution of this dissertation is a command-line tool for equivalence checking of π-calculus models – `pisim22`. The tool supports strong and weak early bisimulation checking. It uses Leung's `pifra` as a front-end and implements the previously presented n-bisimulation algorithm as a key part of the back-end. Implementation required careful choice of data structure to make it efficient. The example of using the tool and details of its implementation were given in chapter 4.

One of the objectives was to investigate the practicality of the chosen approach. This was assessed by testing the created tool's correctness and runtime performance. This evaluation was described in chapter 5, and the results were that the tool was correct on all the test cases checked. These test cases included many well-known and realistic examples from the literature and test sets of similar tools. Time performance was evaluated on the same test cases. It was found that performance was better than predicted in the worst-case time complexity analysis. However, the overall conclusion was that the tool is likely not practical as it struggled on moderately large models. In particular, it was found that `pifra` is often a bottleneck in the created tool. On the positive side, the tool successfully implemented strong and weak early bisimulation. And the correctness of its equivalence checking compared favourably to several other publicly available competitors.

6.2 Reflection

In conclusion, all the set objectives were successfully achieved. The previous attempt at creating an equivalence checker for π-calculus through FRAs was examined. Significant limitations were found, and counter-examples demonstrating the possibility of false negatives and false positives were presented. A different approach within the same framework was taken to address these limitations. As per objectives, an algorithm for n-bisimulation was proposed, and a concrete implementation of an equivalence checker using the algorithm was created. The practicality of the tool was evaluated on a number of test cases. The results confirmed that the approach is correct and viable, but its practicality is limited by its performance on moderately-large models.

6.3 Future work

Any non-trivial work is never perfect nor complete. There are several directions for future work. One concerns improving the solution presented in this dissertation, while the other is about investigating alternative algorithms within the same approach with FRAs.

6.3.1 Further work on `pisim22`

This subsection discusses potential future work that would continue directly from the artefacts from this dissertation.

The current version of the equivalence checker is a command-line tool that takes inputs from files. The tool's usability can be improved by, for example, allowing for inline inputs and adding a graphical user interface (GUI). Some similar tools such as `PiET` and HAL[15] have GUIs. On the other hand, MWB supports an interactive read–eval–print loop (REPL) mode of interaction. Since one potential application of `pisim22` is in the education setting for learning about π-calculus and equivalence checking, a high degree of usability could be important.

Another improvement that would improve usability is adding support for polyadic π-calculus. Currently, the tool uses a mostly unchanged version of `pifra` which only supports monadic π-calculus. That is, only one name can be sent or received atomically over a channel. In the polyadic version, this is extended to zero or more names. A polyadic version would make expressing certain models easier. Moreover, the polyadic π-calculus supports an idea of sorts, which is similar to types in functional programming.[24] Additional improvements to the front-end could be adding support for explicit specification of τ-transitions, allowing for comments in files with π-calculus models, and limiting the scope of equality and inequality prefixes to minimise the number of brackets needed.

Perhaps the most significant deficiency of the implemented equivalence checker is its performance. However, only a limited evaluation of time performance was done. As mentioned in the Evaluation chapter, it would be desirable to have a more rigorous evaluation. In particular, future work could include creating new test cases that would allow estimating time complexity per each input variable. It could also include measuring memory complexity. While memory was not an issue in the performed evaluation due to moderately small models and time limits, it could potentially be a problem if the tool is run on larger inputs. Importantly, it would be beneficial to have a quantitative comparison of performance against the other available tools such as `PiET` or MWB. It would allow concluding whether the FRA approach indeed suffers from excessive state expansion or not. Nevertheless, even in current evaluation it was determined that `pifra` is often a bottleneck in `pisim22`. Future work might include looking at how it can be made more efficient.

A more ambitious future work could look at adding support for model checking to `pisim22`. Model checking is concerned with verifying properties about the models, where properties are usually expressed in modal or temporal logic such as μ-calculus[9] or linear temporal logic (LTL). Both MWB and HAL have some support for model checking. An even more challenging project might investigate the relationship between model checking and n-bisimulation. Specifically, it could be attempted to provide concise reasons explaining why models are not equivalent.

6.3.2 Alternative implementations of bisimulation through FRA

This dissertation presented one particular approach for equivalence checking through FRAs. It is reasonable to expect that alternative approaches are possible.

In particular, in this dissertation, an on-the-fly algorithm for n-bisimulation was presented. Future work might include investigating whether an efficient global algorithm for n-bisimulation can be created. The current algorithm suffers from potential state space explosion. Hence, it would be interesting to investigate whether a more succinct representation of the state space is possible. Perhaps the space can be divided into some equivalence classes, and perhaps ideas from [29] can be extended to the non-deterministic FRAs and n-bisimulation.

It is also conceivable that the implementation can be structured differently. It is currently structured as a sequence of operations. Moving from this modular architecture to a more monolithic one can provide certain benefits. For example, combining translation from π-calculus to FRA and bisimulation checking into a single can allow for finding inequivalences even in some infinite FRAs.

Source Code

All the source code was attached together with the report at the time of submission. Alternatively, the source code, at the time of writing, is hosted on the GitHub account of the author. The source code for the tool `pisim22` details of which were presented in this dissertation can be found at `https://github.com/yungene/pisim22`. The updated version of `pifra` tool can be found at `https://github.com/yungene/pifra`.

Bibliography

[1] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. *ACM Sigplan Notices*, 36(3):104–115, 2001.

[2] Michael Backes, Catalin Hritcu, and Matteo Maffei. Automated verification of remote electronic voting protocols in the applied pi-calculus. In *2008 21st IEEE Computer Security Foundations Symposium*, pages 195–209. IEEE, 2008.

[3] Bruno Blanchet. Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Foundations and Trends in Privacy and Security*, 1(1–2):1–135, Oct. 2016.

[4] Amar Bouali and Robert De Simone. Symbolic bisimulation minimisation. In *International Conference on Computer Aided Verification*, pages 96–108. Springer, 1992.

[5] Vincent Cheval and Bruno Blanchet. Proving more observational equivalences with ProVerif. In *International Conference on Principles of Security and Trust*, pages 226–246. Springer, 2013.

[6] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The Concurrency Workbench. In *International Conference on Computer Aided Verification*, pages 24–37. Springer, 1989.

[7] Rance Cleaveland and Oleg Sokolsky. Equivalence and preorder checking for finite-state systems. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 391–424. Elsevier Science, Amsterdam, 2001.

[8] Basil Contovounesios. An Equivalence Checker for Pi-Calculus Models. Master's dissertation, University of Dublin, Trinity College, 2021.

[9] Mads Dam. Model checking mobile processes. *Information and Computation*, 129(1):35–51, 1996.

[10] Jean-Claude Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13(2-3):219–236, 1990.

[11] Jean-Claude Fernandez and Laurent Mounier. A tool set for deciding behavioral equivalences. In *International Conference on Concurrency Theory*, pages 23–42. Springer, 1991.

[12] Jean-Claude Fernandez and Laurent Mounier. "On the fly" verification of behavioural equivalences and preorders. In *International Conference on Computer Aided Verification*, pages 181–191. Springer, 1991.

[13] Gianluigi Ferrari, Stefania Gnesi, Ugo Montanari, Marco Pistore, and Gioia Ristori. Verifying mobile processes in the HAL environment. In *International Conference on Computer Aided Verification*, pages 511–515. Springer, 1998.

[14] Raffaella Gentilini, Carla Piazza, and Alberto Policriti. From bisimulation to simulation: Coarsest partition problems. *Journal of Automated Reasoning*, 31(1):73–103, 2003.

[15] Stefania Gnesi and Gianluca Trentanni. The HAL-online tool. Technical report, ISTI Technical reports, 2011, 2011.

[16] Michael Kaminski and Nissim Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994.

[17] Paris C Kanellakis and Scott A Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and computation*, 86(1):43–68, 1990.

[18] Seng Leung. Modelling Concurrent Systems: Generation of Labelled Transition Systems of Pi-Calculus Models through the Use of Fresh-Register Automata. Master's dissertation, University of Dublin, Trinity College, 2020.

[19] Huimin Lin. "On-the-fly instantiation" of value-passing processes. In *Formal Description Techniques and Protocol Specification, Testing and Verification*, pages 215–230. Springer, 1998.

[20] Huimin Lin. Computing bisimulations for finite-control π-calculus. *Journal of Computer Science and Technology*, 15(1):1–9, 2000.

[21] Radu Mateescu and Gwen Salaün. Translating Pi-calculus into LOTOS NT. In *International Conference on Integrated Formal Methods*, pages 229–244. Springer, 2010.

[22] Robin Milner. *A Calculus of Communicating Systems*. Springer, 1980.

[23] Robin Milner. *Communication and Concurrency*, volume 84. Prentice Hall Englewood Cliffs, 1989.

[24] Robin Milner. The polyadic π-calculus: a tutorial. *Logic and algebra of specification*, pages 203–246, 1993.

[25] Robin Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, 1999.

[26] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i and ii. *Information and computation*, 100(1):1–40, 1992.

[27] Matteo Mio. PiET - Pi Calculus Equivalences Tester. Available at http://piet.sourceforge.net/ (visited: 2022-04-01), 2006.

[28] Ugo Montanari and Marco Pistore. An introduction to history dependent automata. *Electronic Notes in Theoretical Computer Science*, 10:170–188, 1998.

[29] Andrzej S. Murawski, Steven J. Ramsay, and Nikos Tzevelekos. Polynomial-time equivalence testing for deterministic fresh-register automata. In *43rd International Symposium on Mathematical Foundations of Computer Science (MFCS 2018)*, pages 72:1–72:14. Schloss Dagstuhl, 2018.

[30] Anthony Nash and Sara Kalvala. A framework proposition for cellular locality of dictyostelium modelled in π-calculus. *CoSMoS 2009*, page 83.

[31] Fredrik Orava and Joachim Parrow. An algebraic verification of a mobile network. *Formal aspects of computing*, 4(6):497–543, 1992.

[32] Robert Paige and Robert E Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[33] Frank Puhlmann and Mathias Weske. Using the π-calculus for formalizing workflow patterns. In *International Conference on Business Process Management*, pages 153–168. Springer, 2005.

[34] Davide Sangiorgi. On the origins of bisimulation and coinduction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(4):1–41, 2009.

[35] Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2011.

[36] Davide Sangiorgi and David Walker. *The pi-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[37] Nikos Tzevelekos. Fresh-Register Automata. *ACM SIGPLAN Notices*, 46(1):295–306, Jan. 2011.

[38] Björn Victor and Faron Moller. The Mobility Workbench – a tool for the π-calculus. In *International Conference on Computer Aided Verification*, pages 428–440. Springer, 1994.