An Equivalence Checker for Pi-Calculus Models

Basil Contovounesios

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: Vasileios Koutavas

August 2021
Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Basil Contovounesios

9th August 2021
Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this dissertation upon request.

Basil Contovounesios

9th August 2021
Reactive systems are often formally modelled in one of two ways: in a process calculus such as the π-calculus, in terms of message-passing; or as a labelled transition system (LTS), in terms of a set of states and a transition model. The latter representation is amenable to automata-theoretic algorithmics, but classic formulations cannot effectively capture real-world problems over unbounded domains. This can be mitigated by expressing the LTS of a process as a fresh-register automaton (FRA).

This dissertation builds on prior work in generating the LTS of a π-calculus model represented as an FRA. The aim of the project is to create a model checking tool that adapts known bisimulation algorithms to FRA representations. Using a custom implementation of a classic partition refinement algorithm due to Kanellakis and Smolka, the tool decides whether simple π-calculus models are bisimilar, and if so visualises sets of bisimilar states in the LTS.
Acknowledgements

This dissertation would not have seen the light of day, were it not for the continued support and encouragement of several people.

First and foremost, I would like to thank my supervisor, Vasileios Koutavas, for introducing me to this fascinating topic, as well as for his guidance, advice, and patience throughout the project and previous years of teaching.

Finally, I would like to thank my family, for believing in me and tirelessly pushing me forward every step of the way.

Basil Contovounesios

University of Dublin, Trinity College
August 2021
# Contents

Abstract ........................................ iii

Acknowledgements ................................ iv

List of Figures .................................. vi

List of Listings .................................. vii

Chapter 1 Introduction .......................... 1

1.1 Motivation .................................... 2

1.2 Objectives .................................... 3

1.3 Outline ........................................ 4

Chapter 2 State of the Art ....................... 5

2.1 Background .................................... 5

2.2.1 Bisimulation .................................. 6

2.7.1 Partition-Refinement Algorithms .......... 8

2.14.1 Pi-Calculus .................................. 13

2.16.1 Fresh-Register Automata ................. 16

2.19 Related Projects ............................. 18

2.20 Summary ...................................... 18
<table>
<thead>
<tr>
<th>Chapter 3 Design</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Pifra Tool</td>
<td>19</td>
</tr>
<tr>
<td>3.2 Bisimulation Algorithm</td>
<td>22</td>
</tr>
<tr>
<td>3.3 Pisim Tool</td>
<td>23</td>
</tr>
<tr>
<td>3.4 Summary</td>
<td>24</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 4 Implementation</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 LTS Generation</td>
<td>25</td>
</tr>
<tr>
<td>4.2 LTS Merging</td>
<td>26</td>
</tr>
<tr>
<td>4.3 Deciding Bisimilarity</td>
<td>26</td>
</tr>
<tr>
<td>4.4 Summary</td>
<td>28</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 5 Evaluation</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Visualisation</td>
<td>30</td>
</tr>
<tr>
<td>5.2 Results</td>
<td>31</td>
</tr>
<tr>
<td>5.3 Summary</td>
<td>33</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 6 Conclusion</th>
<th>34</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Discussion</td>
<td>34</td>
</tr>
<tr>
<td>6.2 Achievements</td>
<td>35</td>
</tr>
<tr>
<td>6.3 Challenges &amp; Limitations</td>
<td>35</td>
</tr>
<tr>
<td>6.4 Future Work</td>
<td>36</td>
</tr>
</tbody>
</table>

| Bibliography              | 37 |

| Appendices                | 38 |
List of Figures

2.1 LTS with asymmetrical simulation ......................... 7
2.2 LTS with bisimulation ......................................... 9
2.3 LTS with symmetrical simulation ........................... 10
2.4 Sample LTS with bisimilar states for K&S .................. 12

5.1 Twin LTS annotated with bisimilar states. .................. 32
List of Listings

1  Sample Pifra ASCII output  . . . . . . . . . . . . . . . . . . . . . . . . . 20
2  Sample Pifra DOT output  . . . . . . . . . . . . . . . . . . . . . . . . . 21
3  The split function of Kanellakis & Smolka  . . . . . . . . . . . . . . . . 22
4  The algorithm of Kanellakis & Smolka  . . . . . . . . . . . . . . . . . . . 22
5  Data structures used for Kanellakis and Smolka.  . . . . . . . . . . . . . 27
6  Implementation of Kanellakis and Smolka algorithm.  . . . . . . . . . . . . 28
Chapter 1

Introduction

Historical models of computation such as register machines and the lambda calculus (on which most programming languages are based) capture the computational but not interactional or concurrent behaviour of computing systems [Mil99, p. 3]. The ubiquity of communication and concurrency is not merely a modern trend; they are fundamental behaviours of all computing systems in practice. In fact, each of their computations involves and can be modelled by communication between internal or external components. Formally reasoning about real-world systems is therefore significantly facilitated by a model of computation such as a process calculus, whose basic action is message-passing in the form of concurrent synchronisation over a shared channel.

The $\pi$-calculus in particular was designed for characterising processes with mobile structure, that is the ability to form transient or virtual links with other processes [Mil99, p. 77]. Obvious examples of mobility include connections over wireless networks or between clients and servers on the web, which are continually created and destroyed. While such channels are typically regarded as transmitting static data, the $\pi$-calculus additionally arranges for the transmission of arbitrary processes as messages. This affords the expressive power to elegantly model programming language paradigms [Mil99, p. 113] as well as general computer and security protocols [Bla16].
1.1 Motivation

Designing, implementing, and testing concurrent systems in general, and security protocols in particular, is notoriously difficult and error-prone. New bugs such as race conditions and vulnerabilities to attack are constantly being discovered, to say nothing of those that remain undetected. One approach for reducing these risks is to formally verify the systems, which entails modelling them in a formal language like the $\pi$-calculus [Bla16, p. 2]. It is then possible to reason about all possible behaviours of each system either algebraically, or through generating and analysing its entire state space as an LTS [Leu20, p. 1].

On a theoretical level, models of computation like the $\pi$-calculus describe both the implementation and behavioural specification of processes, meaning a single language can be used for both implementation and formal verification [AIS11, p. 100]. This can mitigate at least to some extent the well-known risk of introducing bugs in the implementation of an otherwise correct specification. As such, it is of fundamental importance and interest to define when different processes are behaviourally equivalent, or even approximately so.

A particularly fine-grained, robust, and well-established notion of equivalence is that of bisimilarity, in which two potentially nondeterministic processes symmetrically simulate each other’s inputs and outputs [PS19]. It can be used to verify whether two processes behave as expected, describe similar behaviour at perhaps different levels of abstraction, or violate certain specified properties, regardless of context or environment [Mil09, pp. xv, 73]. This final point affords greater compositional reasoning, i.e. the ability to infer global properties of an entire system while verifying its components in isolation [BCC98, p. 81]. By contrast, classic model checkers like SPIN do not verify all interactions with the environment, and thus do not compose efficiently in real-world scenarios [Leu20, p. 2].

Although the $\pi$-calculus operates at too low a level to be of practical use in real-
world verification of, say, security protocols, it forms a simple yet expressive basis for application-specific abstractions at a higher level [Mil99, p. 153]. An example of this is the ‘applied’ π-calculus, which extends the original with cryptographic primitives and is used with the ProVerif symbolic protocol verifier [Bla16].

1.2 Objectives

The work in this dissertation predominantly builds on Leung’s dissertation and Pifra tool [Leu20]. That project in turn implements fundamental theory primarily from Milner’s seminal book Communicating and Mobile Systems: the π-Calculus [Mil99], as well as the paper on Fresh-Register Automata by Tzevelekos [Tze11]. Finally, the classic algorithmics of bisimilarity that are implemented in this dissertation are drawn from a survey by Aceto, Ingolfsdottir and Srba [AIS11].

The goals of this dissertation are as follows:

- Adapt and implement the Kanellakis and Smolka [KS90] partition refinement algorithm for deciding bisimilarity to the labelled transition system (LTS) used by the Pifra tool.

- Devise a strategy for merging two Pifra LTSs in a format suitable as input to the Kanellakis and Smolka algorithm.

- Create a model checking tool that applies the algorithm to Pifra LTSs generated from π-calculus models to decide whether they are bisimilar.

- Visualise positive bisimulation results as graphs corresponding to the input LTSs.

- Test and benchmark the algorithm’s implementation.

The implementation of this project is in the Go programming language, for consistency and easier interoperability with the Pifra library and tool.
Future directions of this project would see the implementation of an improved partition refinement algorithm due to Paige and Tarjan [PT87], as well as the more sophisticated definition of bisimulation over $\times\pi$-calculus models given by Tzevelekos.

1.3 Outline

Chapter 2: State of the Art outlines the theoretical foundations and context of the larger research area, including automata, bisimulation, partition refinement, and process algebra theory.

Chapter 3: Design provides high-level overviews of the Pifra tool that forms the basis of this dissertation, the partition refinement algorithm that was implemented, and the general design of the resulting model checking tool.

Chapter 4: Implementation describes in more detail how the Pifra tool was adapted, as well as how the Kanellakis and Smolka algorithm was implemented in the Pisim tool.

Chapter 5: Evaluation demonstrates how the Pisim tool works by presenting a simple proof of concept.

Chapter 6: Conclusion discusses and reflects on the outputs of this dissertation, as well as its limitations and the major challenges that were faced throughout the project.
Chapter 2

State of the Art

This chapter outlines the theoretical context for the project’s goal, which is adapting a classic partition refinement algorithm to π-calculus models represented as fresh-register automata (FRAs). First, important definitions and results in the theories of automata and concurrency are introduced. These include the notion of bisimulation and partition refinement algorithms. The chapter ends with an overview of the theory behind Leung’s Pifra project [Leu20], namely the π-calculus and FRAs, which this dissertation builds on.

2.1 Background

Definition 2.2 Labelled transition system (LTS). [Mil99, p. 16, definition 3.1]. Let $\mathcal{N}$ be an infinite set of names, denoted by lowercase Roman letters, i.e. $a, b, \ldots \in \mathcal{N}$. Similarly, let $\overline{\mathcal{N}} = \{\overline{a} \mid a \in \mathcal{N}\}$ be the set of co-names, where $\mathcal{N}$ and $\overline{\mathcal{N}}$ are assumed disjoint. Finally, let $\mathcal{L} = \mathcal{N} \cup \overline{\mathcal{N}}$ be the set of labels (or actions), denoted by lowercase Greek letters, i.e. $\alpha, \beta, \ldots \in \mathcal{L}$.

A LTS over $\mathcal{L}$ is a pair $(\mathcal{Q}, \mathcal{T})$ consisting of

- a set $\mathcal{Q}$ of states;
• a ternary relation $T \subseteq (Q \times \mathcal{L} \times Q)$, known as the transition relation.

We write $q \xrightarrow{\alpha} q'$, where $q$ and $q'$ are the source and target states of the transition, respectively, when $(q, \alpha, q') \in T$.

Remark. Intuitively, an LTS thus defined is a generalisation of an automaton that lacks distinguished start or accepting states. As such, a single LTS can represent multiple automata.

Remark. Coinciding names and co-names, such as $a$ and $\overline{a}$, represent opposite ends of a mutual, synchronised interaction; a handshake [Mil99, p.27, section 4.2]. A similar notation is used in the $\pi$-calculus described in section 2.14.1.

### 2.2.1 Bisimulation

**Example 2.3 Illustrating language equivalence.** Consider the LTS in figure 2.1. It is clear that the two automata defined by start states $p_0$ and $q_0$ are language-equivalent, i.e. they accept the same language. Nevertheless, committing to either $a$-labelled transition from $p_0$ means that the automaton can no longer simulate the two distinct transitions from $p_1$. Language equivalence is, therefore, an inappropriate notion of equivalence over reactive systems. A more useful equivalence relation over LTSs is that of bisimulation.

**Definition 2.4 Strong simulation.** [Mil99, p. 17, definition 3.3]. Let $(Q, T)$ be an LTS, and let $S$ be a binary relation over $Q$. $S$ is a strong simulation over $(Q, T)$ if, whenever $pSq$,

$$
(p \xrightarrow{\alpha} p') \implies \exists q' \in Q. \left( q \xrightarrow{\alpha} q' \land p'Sq' \right).
$$

If there exists a strong simulation $S$ such that $pSq$, we say that $p$ strongly simulates $q$.

Remark. Intuitively, ‘$p$ simulates $q$’ means that $p$ can match every sequence of transitions in $q$ while maintaining the same set of choices at each stage.
In figure 2.1, $p_0$ strongly simulates $q_0$ under the following simulation $S$:

$$S = \{(q_0, p_0), (q_1, p_1), (q'_1, p_1), (q_2, p_2), (q_3, p_3)\}$$

Indeed, by examining each pair $(q, p) \in S$, we can verify that each transition $q \xrightarrow{\alpha} q'$ of $q$ is matched by some transition $p \xrightarrow{\alpha} p'$ of $p$.

However, the converse is not true: $q_0$ does not strongly simulate $p_0$. In other words, $S^{-1}$ is not a simulation. We now define bisimulation in terms of a strong simulation and its converse.

**Definition 2.5 Strong bisimulation, strong equivalence.** [Mil99, p. 18, definition 3.6]. A binary relation $S$ over $Q$ is said to be a **strong bisimulation** over the LTS $(Q, T)$ if both $S$ and $S^{-1}$ are simulations. If there exists a strong bisimulation $S$ such that $pS q$, $p$ and $q$ are said to be **strongly bisimilar** or **strongly equivalent**, written $p \sim q$. 

---

**Figure 2.1:** LTS with asymmetrical simulation [Mil99, p. 17, example 3.2].
Example 2.6 Illustrating bisimulation. [Mil99, p. 18, example 3.7]. Consider the LTS in figure 2.2a. If we define $S$ as the bisimulation

$$S = \{(p_0, q_0), (p_0, q_2), (p_1, q_1), (p_2, q_1)\}$$

then $p_0 \sim q_0$ and thus $p_0 \sim q_0$. Figure 2.2b further highlights the bisimulation by linking bisimilar states.

Example 2.7 Illustrating nonequivalence. [Mil99, p.20, exercise 3.10]. Consider the LTS in figure 2.3. It is clear that $p_0$ and $q_0$ simulate one another, yet $p_0 \not\sim q_0$. $p_0$ can enter deadlock, which can be simulated by inactivity. Thus bisimulation is a stronger condition than symmetrical simulation.

2.7.1 Partition-Refinement Algorithms

Section 2.2.1 defined bisimilarity over finite LTSs, namely those with finitely many states and transitions. This section will introduce a classic approach to deciding bisimilarity over such systems algorithmically.

Preliminaries

Definition 2.8 Stratified bisimulation relations. [AIS11, p.103, definition 3.2.2]. Given an LTS $(Q, \mathcal{T})$, the stratified bisimulation relations $\sim_k \subseteq Q \times Q$ for $k \in \mathbb{N}$ are defined as

- $E \sim_0 F \quad \forall E, F \in Q$;

- $E \sim_{k+1} F \quad$ iff for each action in $\mathcal{T}$:
  1. $\left(E \xrightarrow{a} E'\right) \rightarrow \exists F' \in Q. \left(F \xrightarrow{a} F' \land E' \sim_k F'\right)$; and
  2. $\left(F \xrightarrow{a} F'\right) \rightarrow \exists E' \in Q. \left(E \xrightarrow{a} E' \land E' \sim_k F'\right)$. 

8
Figure 2.2: LTS with bisimulation [Mil99, p. 19, example 3.7].
Each stratified bisimulation relation $\sim_k$ is also known as the $k$th approximant to bisimilarity. This implies that, for any image-finite LTS, i.e. one whose transitions have a finite set of targets, stratified bisimilarity coincides with bisimilarity for sufficiently large $k$.

**Lemma 2.9.** [AIS11, p.104, lemma 3.2.3]. Given an image-finite LTS $(Q, T)$ with $E, F \in Q$, $E \sim F$ iff $\forall k \in \mathbb{N}. (E \sim_k F)$.

The two definitions above lead naturally to a sketch of an algorithm:

1. Start with the universal relation $\sim_0$;

2. Iteratively compute $\sim_k$ from $\sim_{k-1}$ until convergence.

**Definition 2.10 Partition.** [AIS11, p. 104, section 3.2.1]. Given an LTS $(Q, T)$, a partition $\pi$ of $Q$ is a set of nonempty and disjoint blocks $B_k$, $k \in \mathbb{N}$ such that $Q = \bigcup_k B$.

**Remark.** Intuitively, $\pi$ is used to define an equivalence relation over $Q$.

**Definition 2.11 Refinement.** [AIS11, p. 105, section 3.2.1]. Let $\pi$ and $\pi'$ be partitions of $Q$. $\pi'$ is called a refinement of $\pi$ if $\forall B' \in \pi'. \exists B \in \pi. (B' \subseteq B)$. In that case,
the equivalence relation associated with the refinement $\pi'$ is included in that associated with $\pi$.

These definitions lead naturally to so-called partition-refinement algorithms for computing bisimilarity, which iteratively refine an initial partition until that coincides with the largest strong bisimulation over the input LTS. This method is also referred to as solving the relational coarsest partitioning problem. Two classic algorithms under this banner are due to Kanellakis and Smolka [KS90], and Paige and Tarjan [PT87], the latter being an improvement of the former. This dissertation focuses only on the former.

**Kanellakis & Smolka Algorithm**

The previous section established the notion of partition refinement. The key to a partition-refinement algorithm is the criterion for successively refining each partition. In the case of the Kanellakis and Smolka algorithm, partitions are refined based on so-called splitter blocks.

**Definition 2.12 Splitter.** [AIS11, p.105, definition 3.2.4]. Let $(Q, T)$ be an LTS, and let $\pi = \{B_0, \ldots, B_k\}, k \in \mathbb{N}$, be a partition of $Q$. A splitter for a block $B_i \in \pi$ is a block $B_j \in \pi$ (where it is possible that $i = j$), such that, for some action $a$ in $T$, some states in $B_i$ have $a$-labelled transitions to a state in $B_j$, and others do not.

**Remark.** Intuitively, the presence of a splitter implies that the relevant states are not bisimilar, as they afford different sets of transitions. These differences form the lines along which blocks are successively refined into subsets. This process is repeated until the elements in a block are indistinguishable based on their transitions to other blocks.

**Example 2.13 Illustrating K&S algorithm.** [AIS11, p.106, example 3.2.5]. Consider the LTS $(Q, T)$ in figure 2.4, with the initial partition $\pi = \{Q\} = \{\{s, s_1, s_2, t, t_1\}\}$. 
States in \( Q \) afford either \( a \)- or \( b \)-labelled transitions, so \( Q \) is a splitter unto itself with respect to either transition label. By convention, we process labels in lexicographic order. Thus, splitting \( Q \) with respect to \( a \) gives the refined partition \( \{ \{ s, t \}, \{ s_1, s_2, t_1 \} \} \).

Neither resulting block can be split further with respect to \( a \) or \( b \), so this permutation coincides with the largest strong bisimulation over this LTS. Specifically, states within each block of the final partition are bisimilar to one another.

![Sample LTS with bisimilar states for K&S](image)

Figure 2.4: Sample LTS with bisimilar states for K&S [AIS11, p. 106, example 3.2.5].

**Theorem 2.14 Complexity of K&S algorithm.** [AIS11, p. 109, theorem 3.2.7].

Given a finite LTS with \( n \) states and \( m \) transitions, the algorithm of Kanellakis and Smolka finds the partition corresponding to strong bisimilarity in \( O(nm) \) time.

This complexity result is known to be suboptimal. Indeed, the algorithm of Paige and Tarjan improves on this with a time complexity of \( O(m \log n) \). While relying on the same partition refinement mechanism, the algorithm uses more complex data structures to inform future splits with information from previous splits.

A more fundamental limitation of these algorithms is that they require constructing the entire state-space in advance, which is often impractical when modelling more interesting and complex classes of concurrent systems. Mitigations include algorithms which construct the LTS ‘on-the-fly’ [Lin00]; as well as novel definitions of symbolic models and bisimulations thereon, as will be seen with FRAs in section 2.16.1.
Due to time constraints and the complexity of these approaches, they were ultimately deemed out of scope for this dissertation.

2.14.1 Pi-Calculus

Having established simple, finite LTS models in previous sections, we now introduce a more complex model for realistic systems.

Process calculi are algebras for modelling reactive concurrent systems, particularly in the research area of model checking [Leu20, p. 7]. While such systems can also be modelled by an LTS, a process calculus facilitates symbolic manipulation, and also provides a single, uniform language for expressing both the specification and implementation of a system [AIS11, p. 100].

Influential and heavily studied process calculi include communicating sequential processes (CSP) [Hoa85], the calculus of communicating systems (CCS) [Mil89] and, for the purpose of this dissertation, the $\pi$-calculus [Mil99]. The latter is set apart from other models in particular due to its focus on mobility, in which local data, messages, and entire systems themselves are all treated and transmissible in a uniform way.

While there are many subtly different and application-specific definitions for the $\pi$-calculus, the following definition is a standard one based on [Mil99].

**Definition 2.15 The $\pi$-calculus.** [Mil99, p. 87, definition 9.1; Leu20, p. 7, definition 2.1]. Let lowercase Roman letters $x, y, \ldots \in \mathcal{N}$ represent names as per definition 2.2.
The following set of process expressions defines the abstract syntax of the $\pi$-calculus:

\[
P, Q ::= \quad \text{process} \\
\quad | \ x(y). P \quad \text{input} \\
\quad | \ \overline{x}(y). P \quad \text{output} \\
\quad | \ \tau. P \quad \text{unobservable action} \\
\quad | \ \nu x P \quad \text{restriction} \\
\quad | \ P + Q \quad \text{summation} \\
\quad | \ P | Q \quad \text{composition} \\
\quad | \ !P \quad \text{replication} \\
\quad | \ 0 \quad \text{inaction}
\]

where $P, Q$ stand for any type of $\pi$-calculus process.

The input $x(y)$, output $\overline{x}(y)$, and tau $\tau$ expressions are collectively called the action prefixes of the $\pi$-calculus. They generalise hitherto seen LTS actions, and represent receiving or sending messages (i.e. names), or making a silent, internal transition, respectively. Once their action has been performed, their successor process $P$ can be executed.

The restriction expression $\nu x P$ restricts the scope of the name $x$ to the process $P$, making it one of the bound names in $P$. $P$ may also contain other, unbound names; such names are said to be the free names in $P$.

The summation expression $P + Q$ indicates a nondeterministic, mutually exclusive choice between executing process $P$ or process $Q$. Conversely, the composition $P \mid Q$ executes both processes in parallel.

The replication expression $!P$ generates an unbound stream of copies of the process $P$. This affords recursive process definitions which, for instance, can emulate analogous
function definitions in conventional programming languages.

Finally, the empty, nil, or inaction expression 0 indicates the termination of a process, and is often omitted for brevity.

**Example 2.16 Illustrating $\pi$-calculus reactions.** [Mil99, p. 88, example 9.2]. Consider the process

$$P = \nu z ((\pi(y) + z(w).\overline{w}(y)) \mid x(u).\pi(v) \mid \pi(z)).$$

Note that the inner composition lists processes that begin with unqualified, *complementary* input and output actions, such as $x(u)$ and $\pi(z)$. These readily interact with each other in an event called a *redex*. The *firing* of a redex in turn constitutes the *reaction* $P \rightarrow P'$, where $P'$ substitutes the received name for the locally bound name.

In this example, $x(u)$ is complemented by one of $\pi(y)$ or $\pi(z)$. These redexes give rise to one of possibly two reactions and corresponding substitutions $P \rightarrow P_1$ or $P \rightarrow P_2$:

$$P_1 = \nu z (0 \mid \overline{y}(v) \mid \pi(z))$$

$$P_2 = \nu z ((\pi(y) + z(w).\overline{w}(y)) \mid \pi(v) \mid 0).$$

While $P_1$ cannot react any further, $P_2$ can synchronise on the name $z$ to give rise to the reaction $P_2 \rightarrow P_3$, where

$$P_3 = \nu z (\overline{v}(y) \mid 0 \mid 0).$$

Apart from affording some intuition for the mechanics of the $\pi$-calculus, the example above can also highlight some of the difficulties the $\pi$-calculus can present. Note that the outer restriction in $P$ binds the name $z$ locally. Peeking further into the process, we see that the names $u$ and $w$ are also bound within certain subprocesses, but by
input instead. The rest of the names, however, are left free.

Given the infinite domain of names, this means that $P$ may react in unknown or unpredictable ways when placed in different environments or contexts. With replication as another source of infinity in the $\pi$-calculus, it should be clear that many models cannot readily be represented by the finite kinds of LTSs we have seen so far. The next section introduces a variation of automata and the $\pi$-calculus to mitigate some of these challenges.

### 2.1.6.1 Fresh-Register Automata

The previous section outlined the ways in which $\pi$-calculus models may grow to infinite numbers of states. In this section, we describe a new type of automaton developed by Tzevelekos [Tze11], called a fresh-register automaton (FRA), that can finitely represent many infinite models through the use of auxiliary memory and history.

One of the key concepts to FRAs is that of fresh names, in the sense of [Nee93]. Two examples of local freshness we have already seen in the $\pi$-calculus are restriction $\nu x$ and input $x(y)$, both of which introduce new, previously unseen names to the local scope. Intuitively, this is not that divorced from creating a new object or other resource in a conventional programming language, and in fact this is precisely the type of scenario that research in FRAs aims to tackle. The theory of FRAs also introduces the concept of global freshness, wherein a name is fresh throughout the entire execution of a system.

To capture fresh names, FRAs augment classical automata with both a finite number of registers, each of which can store a name; as well as a history of all names already seen. FRAs can thus compare incoming names with those already stored in a register, and take actions conditional on the comparison.

**Definition 2.17 Fresh-register automaton (FRA).** [Tze11, p. 296, definition 1].

An FRA of $n$ registers is a quintuple $A = (Q, q_0, \sigma_0, \delta, F)$ where:
• $Q$ is a finite set of states,

• $q_0$ is the initial state,

• $\sigma_0 \in \text{Reg}_n$ 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.

$\text{Reg}_n$ is a set of functions, called register assignments, which map one of the $n$ register indices in $\mathcal{A}$ to a unique stored name.

The full transition and history semantics of an FRA are formally defined by encoding states as so-called configurations.

**Definition 2.18 Configuration.** [Tze11, p. 297, definition 2]. A configuration of $\mathcal{A}$ is a triple $(q, \sigma, H) \in \hat{Q}$, with

$$\hat{Q} = Q \times \text{Reg}_n \times \mathcal{P}_{fn}(\mathbb{A})$$

and $\mathcal{P}_{fn}(\mathbb{A})$ being the finite subsets of the infinite set of names $\mathbb{A}$.

Since the basic $\pi$-calculus introduced in section 2.14.1 cannot readily be captured by the FRA semantics described above, Tzevelekos also defines an extended version of the $\pi$-calculus, called the $\times\pi$-calculus [Tze11, p. 302, definition 29]. Its chief deviation from the basic $\pi$-calculus is the introduction of two new process expressions.

The process expression $[a = b]P$ executes $P$ only if the names $a, b$ are equivalent. Meanwhile replication $!P$ is replaced with process definition $p(\vec{a}) = P$, where $\vec{a}$ is a vector of names as arguments, thus coming even closer to the function call semantics in conventional programming languages.
2.19 Related Projects

This dissertation builds directly on the Pifra tool developed as part of Leung’s MCS thesis [Leu20] under the same supervisor. Pifra is a library and command-line tool written in the Go programming language which parses models written in the $\pi$-calculus and generates the corresponding LTS represented as an FRA.

In order to build the LTS, Pifra implements the $\times \pi$-calculus transition relation and applies structural congruence [Mil99, p. 90, definition 9.7] on the resulting states, which are encoded as FRA configurations. The result is a ‘normalised’ LTS in standard form [Mil99, p. 90, definition 9.12]. Intuitively, this means that superficially equivalent LTSs that can easily be transformed into one another are collapsed into a normal form that facilitates analysis. One possible transformation, for example, when two FRAs have identical register contents that differ only in the stored name, is to permute stored names until the register contents trivially match. Such FRAs are said to be structurally congruent.

Using Pifra as a front-end for converting $\pi$-calculus models into a normalised FRA representation, this dissertation then implements and adapts the bisimulation algorithm of Kanellakis and Smolka described in section 2.7.1 on the resulting LTS.

2.20 Summary

This chapter introduced theoretical concepts from automata and process theory that underpin the partition refinement algorithm and Pifra project that this dissertation adapts and builds on.
Chapter 3

Design

This chapter provides a high-level overview of the Pifra tool that this dissertation builds on, the bisimulation algorithm that was implemented, and the major design choices taken to achieve that. Chapter 4 will then describe in more technical detail the implementation of Pisim, the tool created as part of this dissertation.

3.1 Pifra Tool

Section 2.19 gave an overview of the functionality provided by Leung’s Pifra Go library and command-line tool [Leu20], which this dissertation makes direct use of. This section will describe Pifra’s high-level organisation with an eye to reusing its functionality.

As a Go library, Pifra exposes the full parsed and normalised LTS represented by FRAs. Meanwhile, as a command-line utility, it is also able to serialise the LTS in multiple formats, including ASCII plaintext or the GraphViz DOT language [Gra21]. All of these formats contain the full LTS and can thus be used as input for the Pisim equivalence checker.

For example, consider the following π-calculus model [Leu20, p. 59]:
Listing 1 shows the corresponding ‘pretty-printed’ ASCII output that Pifra produces for it. The first line defines the root node and configuration, and the remaining lines each define a labelled transition with source and target states, followed by the target’s state configuration.

\[
\begin{align*}
\text{s0} &= \{(1,#1),(2,#2)\} \mid (\#2(\&2).0 \mid \$\&1.\#1'\&1>.\#2'\&1>.0) \\
\text{s0} &= 2 \mid 1 \mid \text{s1} = \{(1,#1),(2,#2)\} \mid \$\&1.\#1'\&1>.\#2'\&1>.0 \\
\text{s0} &= 2 \mid 2 \mid \text{s1} = \{(1,#1),(2,#2)\} \mid \$\&1.\#1'\&1>.\#2'\&1>.0 \\
\text{s0} &= 2 \mid 3* \mid \text{s1} = \{(1,#1),(2,#2)\} \mid \$\&1.\#1'\&1>.\#2'\&1>.0 \\
\text{s0} &= 1'1^* \mid \text{s2} = \{(1,#1),(2,#2)\} \mid (\#2'\&1>.0 \mid \#2(\&1).0) \\
\text{s1} &= 1'1^* \mid \text{s3} = \{(1,#1),(2,#2)\} \mid \#2'\&1>.0 \\
\text{s2} &= 2'1 \mid \text{s4} = \{(2,#2)\} \mid \#2(\&1).0 \\
\text{s2} &= 2 \mid 3* \mid \text{s3} = \{(1,#1),(2,#2)\} \mid \#2'\&1>.0 \\
\text{s2} &= 2 \mid 2 \mid \text{s4} = \{(1,#1),(2,#2)\} \mid \#2'\&1>.0 \\
\text{s2} &= 2 \mid 3* \mid \text{s5} = \{(1,#1),(2,#2)\} \mid \#2'\&1>.0 \\
\text{s2} &= t \mid \text{s5} = \{\} \mid 0 \\
\text{s3} &= 2'1 \mid \text{s5} = \{\} \mid 0 \\
\text{s4} &= 2 \mid 2 \mid \text{s5} = \{\} \mid 0 \\
\text{s4} &= 2 \mid 1* \mid \text{s5} = \{\} \mid 0
\end{align*}
\]

Listing 1: Sample Pifra ASCII output

Listing 2 shows the same output, but in DOT format.
3.1. PIFRA TOOL  

CHAPTER 3. DESIGN

```plaintext
digraph {
    s0 [peripheries=2,label="{(1,#1),(2,#2)}
        (#2(&2).0 | $&1.#1'<&1>.#2'<&1>.0)"
        s1 [label="{(1,#1),(2,#2)}
            $&1.#1'<&1>.#2'<&1>.0"
            s2 [label="{(1,#1),(2,#2)}
                (#2'<#1>.0 | #2(&1).0)"
                s3 [label="{(1,#1),(2,#2)}
                    #2'<#1>.0"
                    s4 [label="{(2,#2)}
                        #2(&1).0"
                        s5 [label="{}
                        0"]

    s0 -> s1 [label="2 1"]
    s0 -> s1 [label="2 2"]
    s0 -> s1 [label="2 3"]
    s0 -> s2 [label="1' 1"]
    s1 -> s3 [label="1' 1"]
    s2 -> s4 [label="2' 1"]
    s2 -> s3 [label="2 1"]
    s2 -> s3 [label="2 2"]
    s2 -> s3 [label="2 3"]
    s2 -> s5 [label=" "]
    s3 -> s5 [label="2' 1"]
    s4 -> s5 [label="2 2"]
    s4 -> s5 [label="2 1"]
}
```

Listing 2: Sample Pifra DOT output

While the line-oriented ASCII output and standardised DOT format are both readily parsed, for the sake of simplicity and due to time constraints it was decided that Pisim will use Pifra as a library instead, and reuse its well-designed LTS data structure. To facilitate this, Pifra was forked and extended to optionally serialise the LTS in a binary, Go-specific format from which the original data structure can easily be reconstructed.
3.2 Bisimulation Algorithm

This section describes the requirements imposed by the bisimulation algorithm of Kanellakis and Smolka [KS90]. Listings 3 and 4 show the pseudo-code of the algorithm, as defined in [AIS11, p. 108].

\begin{verbatim}
SPLIT(B, a, π)
  1 choose some state \( s \in B \)
  2 \( B_1, B_2 := \emptyset \)
  3 for each state \( t \in B \) do
  4    if \( s \) and \( t \) can reach the same set of blocks in \( π \) via \( a \)-labelled transitions
  5      then \( B_1 := B_1 \cup \{t\} \)
  6      else \( B_2 := B_2 \cup \{t\} \)
  7    if \( B_2 \) is empty then return \( \{B_1\} \)
  8    else return \( \{B_1, B_2\} \)
\end{verbatim}

Listing 3: The subroutine of Kanellakis & Smolka split\((B,a,π)\) [AIS11, p. 108].

\begin{verbatim}
PARTKS(Q)
  1 \( π := \{Q\} \)
  2 changed := true
  3 while changed do
  4    changed := false
  5    for each block \( B \in π \) do
  6        for each action \( a \) do
  7            sort the \( a \)-labelled transitions from states in \( B \)
  8            if split\((B,a,π)\) = \( \{B_1, B_2\} \neq \{B\} \)
  9              then refine \( π \) by replacing \( B \) with \( B_1 \) and \( B_2 \),
          and set changed to true
\end{verbatim}

Listing 4: The algorithm of Kanellakis & Smolka [AIS11, p. 108].

From the algorithm, it is clear that Pisim will need an efficient way of representing sets of states as blocks, and partitions as mutable sets of blocks. In Pifra, states are represented by an integer state ID, and the corresponding FRA configuration. Since
3.3. PISIM TOOL

The output of this dissertation is the Pisim command-line tool. Its purpose is to take as input two π-calculus models, merge them into a single LTS, and apply the Kanellakis and Smolka algorithm for deciding bisimilarity on the result. If the result is negative, Pisim returns with an appropriate error code. Otherwise, it outputs two DOT graphs, similar to those generated by Pifra, one for each input model. The purpose of these graphs is to give a visual and intuitive indication of which states were bisimilar between the two input LTSs. This is achieved by marking each set of bisimilar states with the same node label, and giving a unique label to each block of bisimilar states.

In order to decide bisimilarity, Pisim first merges all the states and transitions from the input LTSs in such a way as to be able to disambiguate the first model and its states from those of the second. The resulting LTS is fed to the Kanellakis and Smolka algorithm, whose output is a partition of states in blocks. Since each block corresponds
to a set of bisimilar states, assessing bisimilarity is reduced to checking that each block contains states from both input LTSs.

### 3.4 Summary

This chapter provided an overview of Leung’s Pifra library and tool, its inputs and outputs, and how they may need to be tweaked in order to allow this dissertation’s Pisim tool to use them most effectively.

The Kanellakis and Smolka algorithm was described, and subsequently analysed to determine what requirements it imposes on the Pisim implementation. Some steps of the algorithm were revised in order to facilitate faithful implementation in the Go programming language.

Finally, the rough flow of data through the Pisim tool was established, from input, and how to disambiguate between two input π-calculus models, to how to interpret the output of the Kanellakis and Smolka algorithm.
Chapter 4

Implementation

This chapter provides a brief overview of the technical aspects involved in implementing the Kanellakis and Smolka bisimulation algorithm using Pifra’s FRA representation, and the extensions to Pifra that were needed to facilitate this.

4.1 LTS Generation

Chapter 3 described the different formats that Leung’s Pifra tool was designed to output. It was decided that re-parsing these outputs would be too time-consuming and would duplicate an unnecessarily large subset of Pifra, so Pisim sought to reuse Pifra’s data structures directly instead. The Go programming language’s gob binary serialisation format was deemed sufficiently simple to use and efficient for this purpose.

Teaching Pifra to output another format was fairly straightforward, thanks to the tool’s clean design. It required adding a new command-line flag and then registering the pertinent LTS data structures with the gob runtime encoder.
4.2 LTS Merging

The Pifra tool takes a single \( \pi \)-calculus model as input, and outputs a single LTS in normal form. In order to apply the Kanellakis and Smolka algorithm on these two systems, they had to be merged into a single LTS. Nevertheless, this had to be done in such a way as to preserve the origin of each state and transition, so as to allow determining from the final partition whether the two systems are indeed bisimilar.

This disambiguation was achieved through a simple pre-processing step over the input FRA LTSs. Since each state is identified by a unique integer, the pre-processing step entailed transforming the state IDs of one LTS into even integers, and those of the other LTS into odd integers.

4.3 Deciding Bisimilarity

Listing 5 shows some of the pertinent Go data structures used for implementing a slightly revised version of the Kanellakis and Smolka algorithm. Specifically, the transition sorting step in the original algorithm, which is meant to guarantee better performance, was instead emulated by mapping states to their containing block.
// States is a set of states.
type States map[int]struct{}

// Actions maps labels to their list of transitions.
type Actions map[pifra.Label][]pifra.Transition

// Block is a set of states, identified by a unique integer.
type Block struct {
    id    int
    states States
}

// Blocks is a set of Blocks keyed by their IDs.
type Blocks map[int]Block

// StateBlocks map State IDs to their containing Block
// for fast lookup.
type StateBlocks map[int]Block

// Partition is primarily a set of Blocks, but also carries some
// auxiliary data to simplify and optimise the implementation.
type Partition struct {
    blocks Blocks
    states StateBlocks
    actions Actions
}

Listing 5: Data structures used for Kanellakis and Smolka.

The algorithm was otherwise faithfully followed, as seen in listing 6.
4.4. SUMMARY

Chapter 4. Implementation

Listing 6: Implementation of Kanellakis and Smolka algorithm.

```go
func partKS(left, right pifra.Lts) Partition {
    part := newPartition(left, right)
    changed := true
    for changed {
        changed = false
        out:
            for id, block := range part.blocks {
                for action := range part.actions {
                    b1, b2 := splitKS(block, action, part)
                    if b1.id == id {
                        continue
                    }
                    refine(part, block, b1, b2)
                    changed = true
                    break out
                }
            }
    }
    return part
}
```

After applying the Kanellakis and Smolka algorithm, the resulting partition was assessed for bisimilarity. This was decided by checking that each block in the partition contains states from both input LTSs.

4.4 Summary

This chapter provided an overview of the main implementation steps involved in applying the Kanellakis and Smolka algorithm, starting with the minor revisions that were made to a fork of Leung’s Pifra library and tool. A description of the merge strategy for the two input LTSs was given, such that their constituent states could still be distinguished in the merged LTS. A code listing demonstrated that the original
algorithm was followed almost entirely faithfully. Finally, the chapter described how the output partition of the algorithm is used for deciding bisimilarity.
Chapter 5

Evaluation

This chapter gives a brief glimpse at how the implementation of the Kanellakis and Smolka algorithm was evaluated.

5.1 Visualisation

The previous chapters described a simple way of analysing the algorithm’s output partition to decide bisimilarity. This section will describe how positive bisimilarity results were visualised.

Since the merge strategy applied to the two input LTSs preserves the respective origin of each set of states, it is easy, after the bisimulation algorithm has returned a positive result, to map bisimilar blocks back to their original LTS. This allows us to traverse the original LTS and mark states from the same block as being bisimilar (in fact, this step is performed concurrently with assessing bisimilarity, to avoid traversing each LTS twice).

The chosen visualisation medium was the GraphViz DOT format thanks to its simplicity, as well as for consistency with the Pifra tool. In the output graphs, each set of bisimilar states is assigned the same state label, which is an integer. Other
visualisations, such as graph colouring, were also considered, but were decided against
given the possibly large number of blocks in the final partition.

5.2 Results

Consider the same sample π-calculus model from section 3.1, written to a file named
fresh.pi:

\[
x.a'\langle x\rangle.b'\langle x\rangle.0 \mid b(y).0
\]

As a sanity check and proof of concept, we specify the same model as both inputs
to the Pisim tool:

\[
\begin{align*}
\$ \text{pifra} \ --\text{output=gob} \ --\text{output} \ fresh.gob \ fresh.pi \\
\$ \text{pisim} \ fresh.gob \ fresh.gob \ out
\end{align*}
\]

The bisimulation result is positive, and the resulting two annotated graphs, one of
which is shown in figure 5.1, are of course equivalent.
Conversely, running Pisim on fresh.pi and the following ping1.pi:

\[
P = a(x).x'\prec x\succ .0 \mid P
\]

\[
P
\]

gives a negative result:

\[
$ pifra --output-gob --output ping1.gob ping1.pi$
\]

\[
$ pisim fresh.gob ping1.gob out$
\]

Not bisimilar

exit status 1

Running Pisim on ping1.pi and the following ping2.pi gives a positive result, but that is due to Pifra’s LTS normalisation to standard form.
This chapter gave a brief glimpse at how Pisim visualises its results, and demonstrated that it works as a proof of concept on obviously equivalent or obviously non-equivalent inputs. Ideally the Pisim tool would have seen more rigorous testing and evaluation. In particular, the original intention was to show that the Kanellakis and Smolka algorithm can correctly identify some, but not most, bisimulations over FRAs, thanks to its insensitivity to configurations. This was unfortunately not possible due to time constraints.
Chapter 6

Conclusion

6.1 Discussion

The original goal of this project was to implement the classical partition refinement algorithms due to Kanellakis and Smolka [KS90], and Paige and Tarjan [PT87], as well as the more complicated definition of bisimulation defined by Tzevelekos over the $\times\pi$-calculus [Tze11], using Leung’s Pifra tool [Leu20] as a starting point.

Due to time constraints and unforeseen circumstances, only the first of those goals was realised, and even then to a very limited extent. In particular, while the implementation is thought to be a faithful adaptation of the Kanellakis and Smolka algorithm, it has not been tested or evaluated to a satisfactory degree to be confident in its correctness. The effective complexity and performance characteristics of the implementation remain unknown for the same reason.

Nevertheless, if this approach is shown to be sound, this project could be used as a basis for implementing the similar but more complex partition refinement algorithm due to Paige and Tarjan. These approaches could further be benchmarked against existing bisimulation tool suites, or any implementations of $\times\pi$-calculus bisimulation. Ultimately these are fundamental and actively researched questions and results in con-
currency and programming language theory, and are ever as relevant today.

6.2 Achievements

The first goal of this project was to adapt the classic Kanellakis and Smolka algorithm to an FRA representation of π-calculus models. While the implementation was realised, there is little confidence in the results as they were not verified for correctness or performance characteristics. Nevertheless, to our knowledge the approach taken is a novel adaptation of a classic algorithm. While the algorithm itself was discovered many decades ago, the theory of FRAs and the formal definition of bisimulation over them was introduced in 2011, and Leung’s Pifra tool for generating the FRA LTS from a π-calculus model was created in 2020.

6.3 Challenges & Limitations

Two main technical challenges were faced while implementing this project. The first was finding a suitable way to capture the output of the Pifra tool. We were originally undecided on how best to communicate between the two tools. The options included linking against Pifra as a library; forking the project; or using it as a command-line program and parsing its output. Initial attempts tried all of the options, and while the command-line tool would have been the cleanest approach, ultimately linking and reusing Pifra’s library proved to be the simplest option.

The second challenge was finding a suitable way to merge two input LTSs so as to make them compatible with the partition refinement algorithm while still being able to distinguish states from each LTS. Ultimately this was solved by devising a simple state ID encoding based on even and odd integers. While it works sufficiently well in this specific case, it may not generalise to more complicated problems or representations.
6.4. FUTURE WORK

The main limitation of this work is that it achieved only a very limited scope. Furthermore, there will be a lack of confidence in its results until they are verified more rigorously. Even if the implementation is shown to be sound, the partition refinement algorithms themselves are known to be too simple to capture bisimulation over the $\times\pi$-calculus. As such, the project offers limited practical value in its current state.

6.4 Future Work

Currently, there is no testing or benchmarking of the Kanellakis and Smolka implementation. The next stage of any similar project would involve devising and collecting a set of tests and benchmarks with which to verify and evaluate the equivalence tool. In particular, it would be interesting to see whether the current or any variant adaptation of the algorithm meets its theorised complexity. With these securities and infrastructure in place, the two partition refinement algorithms could be usefully compared.

Though the partition refinement algorithms are too simple to reliably operate over FRAs, future research could explore more sophisticated ways of adapting them to the larger problem.

Nevertheless, the ultimate goal of the overarching project is to implement bisimulation over the $\times\pi$-calculus as defined in [Tze11]. How some of its more subtle conditions can be implemented in practice remains to be seen.
Bibliography


Source Code

The sources for the implementation described in chapter 4, as well as for this report, can be found at https://gitlab.com/basil-conto/pisim. The extended fork of Leung’s Pifra tool can be found at https://github.com/basil-conto/pifra.