University of Dublin

TRINITY COLLEGE

Modelling Concurrent Systems:
Generation of Labelled Transition Systems of Pi-Calculus Models through the Use of Fresh-Register Automata

Seng Leung

Supervisor: Dr. Vasileios Koutavas

A dissertation

submitted to the University of Dublin, Trinity College
in partial fulfilment of the requirements for the degree of

Master in Computer Science

April 2020

Declaration

I hereby declare that the following dissertation, except where otherwise stated, is entirely my own work; that it has not previously been submitted as an exercise for a degree, either in Trinity College Dublin, or in any other University; and that the library may lend or copy it or any part thereof on request.

Seng Leung
30 April, 2020

Abstract

Concurrent systems are formally modelled by using process calculi, such as the pi-calculus. All possible interactions of a concurrent system with its environment are considered by generating the model's state space, which consists of a set of states and a set of communication transitions called the labelled transition system (LTS). The number of states in an LTS can grow very rapidly and may be infinite, and so ordinary LTS representations are unsatisfactory. Using the paradigm of fresh-register automata, many supposedly infinite-state models can be represented finitely.

This dissertation investigates the generation of LTS's from pi-calculus models through the use of fresh-register automata and implements a tool which does this. From the LTS generated, simple model checking can be performed, which includes verifying if a certain state can be reached. The LTS generated also lays the groundwork for further verification techniques like bisimulation and modal logic assertions.

Acknowledgements

I would like to express my sincere gratitude to my supervisor Vasileios Koutavas for his excellent mentorship, advice, and direction during the course of this dissertation. My discussions with Vasileios always ended with great clarity and insight, and inspired new ideas for the project. The completion of this dissertation would not have been possible without his guidance.

I would also like to thank my parents, who have been there for me throughout my studies providing support and encouragement.

Contents

Introduction 1

Concurrent systems are ubiquitous. Whether it's devices communicating across a network, or programs passing messages around, many systems exhibit concurrency of some form. This dynamic exchange of information was remarked by Milner [1, p. x], who described this as the advent of *mobile computing*, where, "people, computers and software now continually move among each other", and that, "some of the movement is physical and some (e.g. the movement of links) is virtual". This was written in 1999 and his statements are evidenced by today's billions of computing devices communicating with each other in both hardware and software. As a result, the need to understand communication and concurrency is apparent.

The most fundamental way to understanding these mobile systems is through modelling. Modelling a concurrent system is performed through a formal language, like the pi-calculus. From this model, we can consider all possible interactions arising from concurrency and generate the model's state space. This consists of a directed graph of a set of states and a set of transitions, which is called the labelled transition system, or LTS. The generation of LTS's from pi-calculus models is the aim of this dissertation, motivated by model checking in formal verification.

1.1 Motivation

The complexity of software in massively distributed and concurrent systems, such as content storage, load balancing, and search engines, is immense. Suppose someone wanted to make a change to a part of the system, say to improve efficiency. Most commonly, unintended side-effects are captured by regression testing, i.e., by re-running simple input-output tests. However, this does not capture all behaviour. Instead, we could use regression verification where only the change made is inspected to ensure that it respects the previous behaviour. This is the basis of compositional reasoning [2], which reasons about a small part to compose onto a larger system. All interactions with the small part of a system could be exhaustively checked to ensure that it is equivalent to the former behaviour.

There is a growing need for ensuring that security-critical applications are securely built. Consider a system where secret keys are generated and an attacker attempts to access the system. We are not able to know exactly what the attacker would do but we can consider all possible interactions with the system. Suppose that we observe a target state of the system which should not occur under normal circumstances, e.g., nobody but the system should know the secret key and the system receives an existing secret key. If we

simulate the system and find that this state was reached, it shows that this is a possible behaviour in the system. Following the trace leading up to that state reconstructs the attacker's potential actions. This helps uncover underlying bugs or vulnerabilities within the program. Existing model checking tools like *SPIN* [3] do not examine all possible interactions with the context, which makes it unsuitable for compositional reasoning.

1.1.1 Motivating example: Equivalence

As a motivating example for equivalence, consider the two *Go* functions below.

Listing 1.1: Ping server 1.

```
func ping1() {
    go ping1()

    message := <-channel
    channel <- message
}
```

Listing 1.2: Ping server 2.

```
func ping2() {
    message := <-channel

    go func() {
        channel <- message
    }()

    go ping2()
}
```

Both functions attempt to emulate a ping server. A sender sends a message to the receiver (or server) and the receiver sends back the same message.

The following describes succinctly the *Go* syntax: `go ping1()` invokes a concurrent thread to execute `ping1()`. `message := <-channel` receives a message on the channel. The variable `channel` is a global variable while `message` is a local variable known only to the function scope. `channel <- message` sends the same message back on the same channel. `go func() {}()` invokes a thread with an anonymous function.

From inspection, these two seemingly different functions perform the same set of actions. If one were to replace either function with each other, equivalent behaviour would result. The ability to detect for equivalent behaviours would prove useful for swapping components in a program.

To check this, for each of these two functions, the set of actions could be represented by a graph of states and action transitions, i.e., an LTS. Then, from these LTS's, we can assess their equivalence using a notion called bisimulation. This is the ultimate goal of the broader research project. However, for this dissertation, its scope only extends to generating the LTS and performing some verification.

1.1.2 Motivating example: Verification

As a motivating example for verification, consider the following *Go* program.

```
func clientA(as chan chan string) {
    ab := make(chan string)
    as <- ab
    ab <- "hello"
}

func clientB(sb chan chan string) {
    chnl := <-sb
    msg := <-chnl
}

func serverS(as chan chan string, sb chan chan string) {
    chnl := <-as
    sb <- chnl
}

func main() {
    as := make(chan chan string)
    sb := make(chan chan string)
    go clientA(as)
    go clientB(sb)
    go serverS(as, sb)
}
```

Listing 1.3: Communication between client A, client B, and server S.

A message is sent from client A to client B through an intermediate server S. A creates a new channel `ab` and sends this to S, who then forwards it to B. Using this channel, A sends the message `"hello"` to B.

The above system will always result in B receiving the message `"hello"`. This is because the communication channels `as` and `sb` created in `main()` are restricted for use in A, B, and S exclusively. Suppose that these channels are not restricted and are declared as global variables. Another client C could emulate A's actions by creating a channel `cb`, sending it to S, and changing the message to something else, say `"olleh"`. B would then receive `"olleh"` through the channel `cb`. This would violate our assumption that B will always receive the message `"hello"`. An assumption of a system which states that something will not happen is called a *safety* property, whereas something that must occur in a system is called a *liveness* property [4, p. 125].

We can verify the property that B always receives the message `"hello"`. This can be encoded by adding the assertion that `msg` $\neq$ `"hello"`. In generating the LTS of the above system, if the assertion is ever tripped, then we can conclude that our property has been violated. Therefore, finding this state in the LTS would constitute to disproving this property. Likewise, the absence of this state concludes that the system satisfies this property (assuming the LTS is finite). We will model this example using the pi-calculus and formally verify this property in a later chapter.

1.2 Dissertation objectives

The work set out in this dissertation is based primarily on the paper *Fresh-Register Automata* by Tzevelekos [5], and more fundamentally on the pi-calculus subject, Milner's seminal book *Communicating and Mobile Systems: The Pi-Calculus* [1].

The main objective is to generate an LTS from pi-calculus expressions for use in model checking. This pi-calculus language is defined by Tzevelekos in his paper, called the $\times\pi$-calculus. The transition rules, or transition relation, of this language is also defined, which is represented by fresh-register automata. After repeatedly applying the rules, a set of states are obtained, which could be reduced by applying congruence rules, i.e., finding equivalent states. Finally, the LTS is generated which can be used for verification.

In summary, the objectives are as follows.

1. Build a parser for the $\times\pi$-calculus language.
2. Implement the $\times\pi$-calculus transition relation.
3. Apply congruence on the states.
4. Generate and output the LTS.
5. Verify properties of pi-calculus models.

The programming language of choice for implementing this is *Go*.

The next step of this project would be bisimulation, which assesses the equivalence of two LTS's. Although this feature is not implemented, we will explore the foundations of it as it forms one of the primary motivations behind this dissertation.

1.3 Dissertation structure

Chapter 2: Background describes the general related work and the fundamental background of the pi-calculus, bisimulation, and fresh-register automata.

Chapter 3: Language examines how the $\times\pi$-calculus is translated into a language, parsed, and represented internally.

Chapter 4: Transition Relation explores the $\times\pi$-calculus transition relation and its redefined rules for practical implementation.

Chapter 5: Congruence details the application of congruence rules and normalisations to find equivalent states.

Chapter 6: Labelled Transition System outlines the generation and output of the LTS.

Chapter 7: Verification illustrates examples of pi-calculus models, generates their LTS and verifies their properties.

Chapter 8: Evaluation assesses the LTS output and analyses the program performance.

Chapter 9: Conclusion discusses the achievements, challenges encountered, and future work.

Background 2

2.1 General related work

Formal verification is the act of proving or disproving the correctness of a system's behaviour with respect to a formal specification. In a concurrent system, the specification may require the absence of deadlocked processes (which wait indefinitely) or that undesirable states (like system crashes) cannot be reached. The general approaches to formal verification are model checking and deductive verification.

2.1.1 Model checking

Model checking is a method to check whether a model of a system fulfils a given specification. For finite-state models, all states can be exhaustively explored. For infinite-state models, we can only explore up to a certain number of states. However, some can be represented finitely by using abstraction or by a symbolic representation. The properties of a model can be verified by the use of a logic. An example of a model checking tool is *SPIN* [3], which uses *Promela* as the modelling language and linear temporal logic (LTL) as its properties language. The main advantage of model checking is that it is often fully automatic and that it requires little-to-no user intervention. However, its primary setback is that it suffers from the combinatorial blowup of the state space, commonly known as the state explosion problem [6]. The work carried out in this dissertation is based on model checking of process LTS's, which explores the state space of the process but also all possible interactions with its environment.

2.1.2 Deductive verification

Another approach to formal verification is deductive verification. From the system and its specifications, a collection of correctness proofs are constructed which are then fed into a theorem prover. Such proof assistants include *Coq* [7] and *Isabelle* [8]. Hoare logic (HL) [9] forms the basis of deductive verification techniques, which reasons about the preconditions and postconditions of a line of code. Rely-guarantee reasoning [10] is an extension to Hoare logic for reasoning about concurrency. Deductive verification requires the user to understand precisely why the system works correctly (e.g., finding the invariant of a loop) and to convey this information in the verification system.

2.1.3 Process calculi

A process calculus, or process algebra, provides a language to model concurrent systems and can be used to perform both model checking and deductive verification (although it is mostly used for the former). Process calculi provide sets of rules which can be analysed and manipulated to reason about the relationships between processes. Significant examples of process calculi include communicating sequential processes (CSP) by Hoare [11], which influenced *Go* [12], and the calculus of communicating systems (CCS) [13]. For this dissertation, we use the pi-calculus (written as the π-calculus). The π-calculus is based on the CCS, which were both developed by Milner [1, p. 154].

In model checking, the LTS of a model can be generated from the use of process calculi. Properties languages like Hennessy-Milner logic (HML) [14] can be used to specify the properties of the LTS, which can then verify the model. This logic has been formalised for the CCS [15], but there is a consensus that it also works for the π-calculus. Over the last decades, the area of process calculi for use in formal verification has been the focus of research.

2.2 The pi-calculus

There are many variations and slight differences in the definition of the π-calculus, depending on the author or paper context. The π-calculus formalisations to be described are based on Milner's original book *Communicating and Mobile Systems: The Pi-Calculus* [1].

Definition 2.1. The π-calculus. [1, p. 87, definition 9.1]. Let $\mathcal{N}$ be an infinite set of objects called names. Names are usually lower case, i.e., $x, y, z, ... \in \mathcal{N}$. The abstract syntax for the π-calculus is defined by the following:

P, Q	::=	process
	$x(y).P$	input
	$\bar{x}\langle y\rangle.P$	output
	$\nu x\, P$	restriction
	$P + Q$	summation
	$P \mid Q$	composition
	$!P$	replication
	0	inaction

P and Q represent processes. $x(y)$ and $\bar{x}\langle y\rangle$ are defined as action prefixes π, representing either receiving or sending a message (a name) through a channel.

Input

$x(y).P$ receives the name y on channel x, and then runs process P. The received name is bound to y. Binding results in the substitution of y by the name. The input of a name and subsequent binding of that name eliminates, or collapses, the action prefix.

Example 2.1. Input. Suppose we have the expression $x(y).y(z).P$. Upon receiving the name w on channel x, the resultant expression is $w(z).P$. Note the substitution of the bound name y by w, i.e., $\{^w/_y\}$, and the disappearance of the $x(y)$ term.

Output

$\bar{x}\langle y\rangle.P$ sends y on channel x, and then runs process P. As with input, the sending of the name eliminates the action prefix. Input and output complement each other, and gives rise to communication between processes.

Note the overline $\bar{x}$ above the name x. This is called a co-name. Co-names are defined as the set $\bar{\mathcal{N}} = \{\bar{a} \mid a \in \mathcal{N}\}$, complementing the name set $\mathcal{N}$.

Example 2.2. Output. Given the expression $\bar{x}\langle w\rangle.P$, w is sent to an input channel x, which reduces the expression to only P.

Restriction

$\nu x\, P$, also denoted as $\mathsf{new}\, x\, P$, restricts the scope of the name x to the process P. In a restriction $\nu x\, P$, x is a *bound* name. Bound names are local to the scope of the restriction. Names that are not bound in P are called *free* names.

Example 2.3. Restriction. Consider the expression $\nu x\,(y(z).\bar{x}\langle w\rangle.0)$. The bound names are x and z. The name x is bounded by restriction. The name z is bounded by the input action $y(z)$. The free names are y and w, which can be interacted with other processes.

Summation

$P + Q$ represents a nondeterministic choice. Either process P is run or process Q is run, but never both.

Example 2.4. Summation. Consider the expression $\nu x\,(x(y).\bar{y}\langle z\rangle.0 + \bar{x}\langle b\rangle.c(d).0)$. We run exclusively either $x(y).\bar{y}\langle z\rangle.0$ or $\bar{x}\langle b\rangle.c(d).0$. As a result, in this case, these two processes will never communicate.

Composition

$P \mid Q$ means that processes P and Q run in parallel.

Example 2.5. Composition. Consider again the previous expression but summation is now composition $\nu x\,(x(y).\bar{y}\langle z\rangle.0 \mid \bar{x}\langle b\rangle.c(d).0)$. Since the two processes run concurrently, output $\bar{x}$ communicates with input x. The name b is sent over and the expression is reduced to $\nu x\,(\bar{b}\langle z\rangle.0 \mid c(d).0)$.

Replication

$!P$ continuously creates a new copy of the process P an unlimited number of times. This makes the process P persistent. Replication provides the foundation for defining process definitions $A \stackrel{\text{def}}{=} P$, similar to function definitions in programming languages.

Inaction

0 is the nil or inactive process, marking the end of a process. Often, .0 is omitted in expressions for the sake of convenience.

2.2.1 The pi-calculus: Examples

The π-calculus and its related notions are best described by the use of examples.

Example 2.6. Reactions. [1, p. 88, example 9.2]. Consider the expression.

$$P = \nu z\left((\bar{x}\langle y\rangle + z(w).\bar{w}\langle y\rangle) \mid x(u).\bar{u}\langle v\rangle \mid \bar{x}\langle z\rangle\right)$$

The bound names in P are z (by restriction) and, w and u (by input). The free names are therefore x, y and v. Complementary channels input and output of the same name (names and co-names) can interact with each other. This is called a *redex*. The firing of a redex constitutes to a *reaction* $P \rightarrow P'$.

In P, there are two redexes, the pairs $\bar{x}\langle y\rangle$, $x(u)$ and $\bar{x}\langle z\rangle$, $x(u)$. From this expression, there are two possible reactions $P \rightarrow P_1$ and $P \rightarrow P_2$. The reactions invoke a substitution $\{^{y}/_{u}\}$ and $\{^{z}/_{u}\}$, respectively.

$$P_1 = \nu z\,(0 \mid \bar{y}\langle v\rangle \mid \bar{x}\langle z\rangle)$$
$$P_2 = \nu z\,((\bar{x}\langle y\rangle + z(w).\bar{w}\langle y\rangle) \mid \bar{z}\langle v\rangle \mid 0)$$

There are no further redexes in P_1. However, there is one in P_2 – the pair $z(w), \bar{z}\langle v\rangle$. It arose by substitution invoked by the first reaction. As a result, we have $P_2 \rightarrow P_3$.

$$P_3 = \nu z\,(\bar{v}\langle y\rangle \mid 0 \mid 0)$$

From the example, we see that there are a number of reactions. For each redex, the resultant process can be viewed as a state. In the example above, the processes P, P_1,

P_2, and P_3 are states and the redex taken to that state is a transition. In essence, this forms a labelled transition system.

Definition 2.2. Labelled transition system (LTS). [1, p. 16, definition 3.1]. An LTS over a set of actions $\mathcal{L}$ is a pair $(\mathcal{Q}, \mathcal{T})$ consisting of

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

The set of actions, or labels, $\mathcal{L}$ is defined by the union $\mathcal{N} \cup \bar{\mathcal{N}}$. If $(q, \alpha, q') \in \mathcal{T}$, the source state q transitions to the target state q' through action α, i.e., $q \xrightarrow{\alpha} q'$.

Example 2.7. Reactions with the context. We revisit the previous example with the inclusion of the environment, or the context. In the previous example, we were only concerned exclusively with the expression given, i.e., the expression was the entire system. However, we cannot guarantee that the above transitions are the exhaustive list of states if the expression is placed in a wider context or larger system. Therefore, we need to take into account *all* possible names in $\mathcal{N}$ (which is infinite).

We begin from P again. As before, we have two possible transitions to P_1 and P_2. However, because x is not bound, the input $x(u)$ and outputs $\bar{x}\langle y\rangle$ and $\bar{x}\langle z\rangle$ may be reacted with in unpredictable ways. For example, $x(u)$ may receive a free name α that is not u or w, i.e., $\alpha \notin \{u, w\}$, $\alpha \in \mathcal{N}$. In addition, the outputs $\bar{x}\langle y\rangle$ and $\bar{x}\langle z\rangle$ may interact with other process with the channel name x.

For P_2, note that the name z is bounded. Therefore, this name cannot be interacted with outside of the process. However, $\bar{x}\langle y\rangle$ can still react with outside processes as before. We can already see that with the context, the number of additional states can grow very quickly, and possibly infinitely due to the infinite number of names $\mathcal{N}$. An LTS diagram captures the additional states, represented by $P_a, P_b, ..., P_g$. The dots ... represent possible continuations of branches of that state.

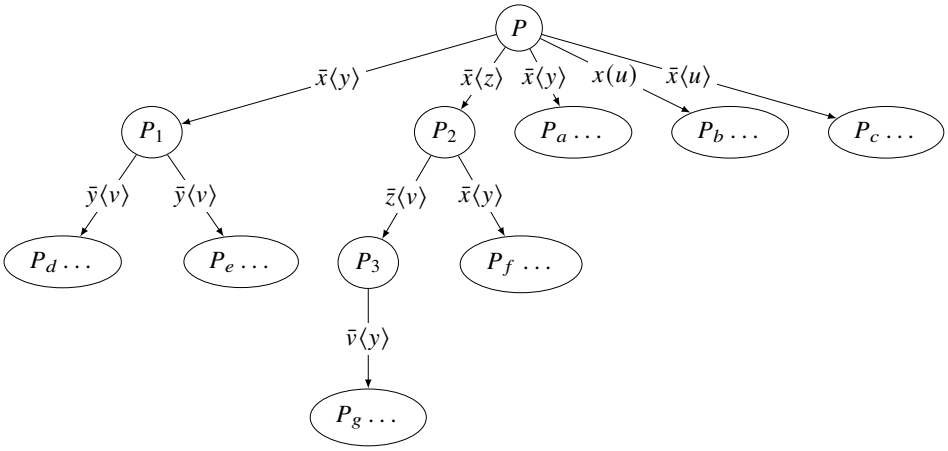


Fig. 2.1: Labelled transition system with reactions from the context.

Example 2.8. Auction. The π-calculus yields expressive capabilities in modelling real-world scenarios. Consider three processes – an auction item a, a bidder y and a bidder z.

$$P = \nu y \left(\bar{a}\langle y\rangle.\bar{y}\langle y\rangle\right) \mid a(x).x(w).\bar{w}\langle w\rangle \mid \nu z \left(\bar{a}\langle z\rangle.\bar{z}\langle z\rangle\right)$$

The bidders y and z both bid on the auction item a. Whichever bidder first communicates with the auction item a will leave the other bidder unable to obtain the item. Because of nondeterminism, the expression can be reduced when either bidder y or when bidder z communicates first with the auction item a.

$$P_{y1} = \nu y \left(\bar{y}\langle y\rangle \mid y(w).\bar{w}\langle w\rangle\right) \mid \nu z \left(\bar{a}\langle z\rangle.\bar{z}\langle z\rangle\right)$$
$$P_{y2} = \nu y \left(0 \mid \bar{y}\langle y\rangle\right) \mid \nu z \left(\bar{a}\langle z\rangle.\bar{z}\langle z\rangle\right)$$

In P_{y1}, y's process locks on with the communication channel a and sends its name y over. Because y is bounded, the binding is extended to a's process. y sends its name over to the auction item's process again, reducing the expression to P_{y2}. Channel y now broadcasts its same name. z's process is presumably deadlocked, depending on the context. Conversely, the same occurs for z's process.

$$P_{z1} = \nu y \left(\bar{a}\langle y\rangle.\bar{y}\langle y\rangle\right) \mid \nu z \left(z(w).\bar{w}\langle w\rangle \mid \bar{z}\langle z\rangle\right)$$
$$P_{z2} = \nu y \left(\bar{a}\langle y\rangle.\bar{y}\langle y\rangle\right) \mid \nu z \left(\bar{z}\langle z\rangle \mid 0\right)$$

The following is the LTS.

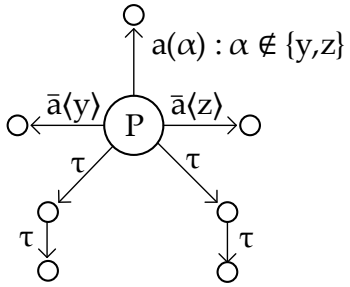


Fig. 2.2: Labelled transition system of the auction model.

Tau steps τ are used to denote unobservable actions, or internal actions. In this case, there are the two τ branches representing the two reactions to P_{y1} and P_{z1}. The second τ step of each are the further reactions to P_{y2} and P_{z2}. Because the auction item a is free, its channel can be communicated from both process P and the context, which is represented by a free name α. This transition describes a scenario where channel a receives a name other than y or z from another process in the context. Alternatively, y or z's process may send their name to another channel a.

This small example demonstrates the powerful construct of name scoping and name passing in the π-calculus. This can also be used as the basis for modelling cryptographic communication protocols. Abadi and Gordon [16] developed an extension to the pi-calculus called the 'spi calculus' for this purpose.

Example 2.9. Hand-over protocol. [1, p. 80]. In a larger real-world example, consider two transmitters and a moving car engaging in a hand-over protocol.

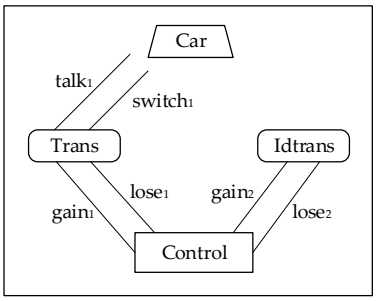


Fig. 2.3: Diagram of the hand-over protocol.

A system with two transmitters is managed by a central controller and a moving car connects to either transmitter. The following expressions represent the system.

$$Car(talk, switch) \stackrel{\text{def}}{=} \overline{talk}.Car\langle talk, switch\rangle + switch(t, s).Car\langle t, s\rangle$$

$$\begin{aligned} Trans_1(talk_1, switch_1, gain_1, lose_1) &\stackrel{\text{def}}{=} talk_1.Trans_1\langle talk_1, switch_1, gain_1, lose_1\rangle \\ &\quad + lose_1(t, s).\overline{switch_1}\langle t, s\rangle.Idtrans_1\langle gain_1, lose_1\rangle \\ Idtrans_1(gain_1, lose_1) &\stackrel{\text{def}}{=} gain_1(t, s).Trans_1\langle t, s, gain_1, lose_1\rangle \end{aligned}$$

$$\begin{aligned} Trans_2(talk_2, switch_2, gain_2, lose_2) &\stackrel{\text{def}}{=} talk_2.Trans_2\langle talk_2, switch_2, gain_2, lose_2\rangle \\ &\quad + lose_2(t, s).\overline{switch_2}\langle t, s\rangle.Idtrans_2\langle gain_2, lose_2\rangle \\ Idtrans_2(gain_2, lose_2) &\stackrel{\text{def}}{=} gain_2(t, s).Trans_2\langle t, s, gain_2, lose_2\rangle \end{aligned}$$

$$\begin{aligned} Control_1 &\stackrel{\text{def}}{=} \overline{lose_1}\langle talk_2, switch_2\rangle.\overline{gain_2}\langle talk_2, switch_2\rangle.Control_2 \\ Control_2 &\stackrel{\text{def}}{=} \overline{lose_2}\langle talk_1, switch_1\rangle.\overline{gain_1}\langle talk_1, switch_1\rangle.Control_1 \end{aligned}$$

The system is initialised with the expression

$$\begin{aligned} System_1 \stackrel{\text{def}}{=} \nu\, & talk_1, switch_1, gain_1, lose_1, talk_2, switch_2, gain_2, lose_2 \\ & (Car\langle talk_1, switch_1\rangle \mid Trans_1 \mid Idtrans_2 \mid Control_1) \end{aligned}$$

The car continually *talks* with one transmitter, until it is notified to *switch* transmitters. The central controller tells one transmitter to *lose* the car, so the transmitter tells the car to *switch*. The transmitter becomes idle. Consequently, the central controller tells the second transmitter to *gain* a car.

Note that $\overline{talk}$ and $talk$ is syntactic sugar for sending/receiving the same name on the channel, i.e., this is short for $\overline{talk}\langle talk\rangle$ and $talk(talk)$.

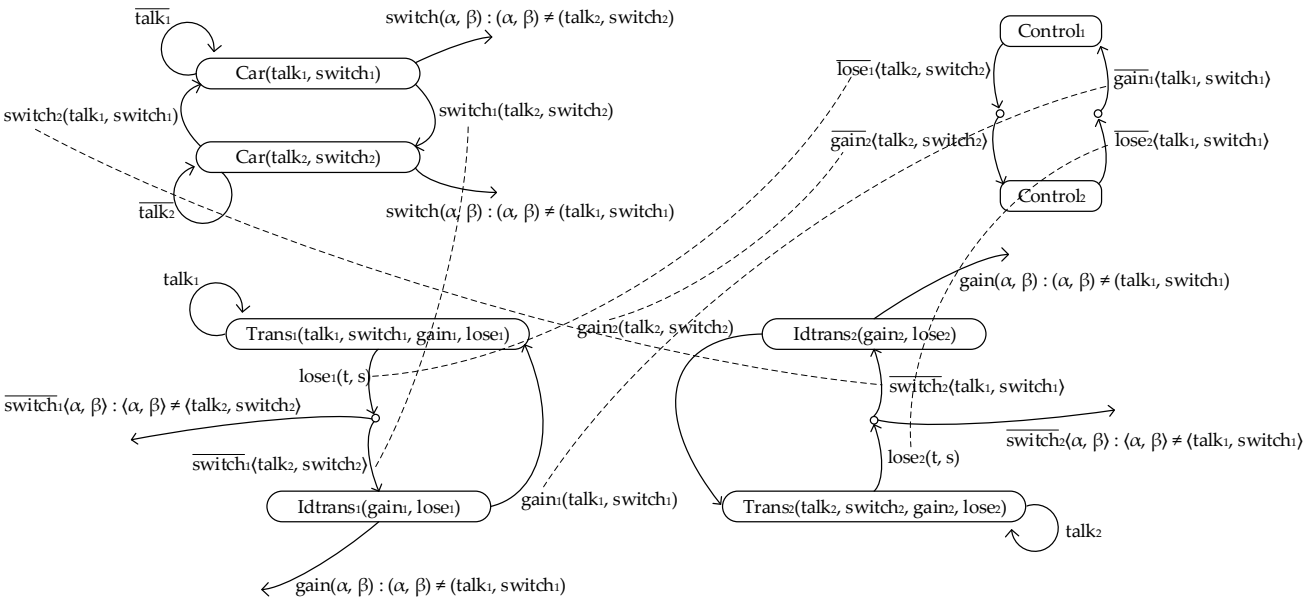


Fig. 2.4: Processes of the hand-over protocol model.

The above is an informal representation of the processes of the system. The dashed lines represent communication channels between the processes. Note the transitions where the names are not defined in the expressions which lead to undefined behaviours, e.g., $\text{switch}(\alpha, \beta) : (\alpha, \beta) \neq (\text{talk}_2, \text{switch}_2)$. These transitions represent the behaviour of all the other names in the context. This captures the full behaviour of the process when placed in any context. Each process at a given state relative to the other processes in the system would constitute an overall state in the LTS of the model.

2.3 Bisimulation

To check for equivalence of LTS's, we use the notion of bisimulation. Bisimulations check if one system simulates the other and vice versa.

2.3.1 Strong simulation

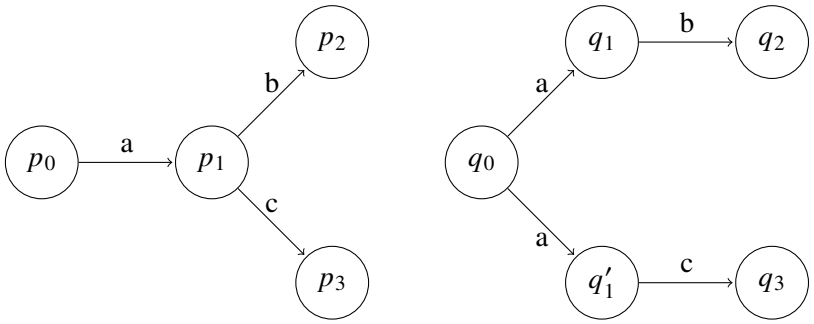
We consider a notion of equivalence between two states in an LTS called simulation.

Definition 2.3. Strong simulation. [1, p. 17, definition 3.3]. Let $(\mathcal{Q}, \mathcal{T})$ be an LTS. Let $\mathcal{S}$ be a binary relation over $\mathcal{Q}$. $\mathcal{S}$ is called a strong simulation over $(\mathcal{Q}, \mathcal{T})$ if whenever $p\mathcal{S}q$,

if $p \xrightarrow{\alpha} p'$, then $\exists q' \in \mathcal{Q} : q \xrightarrow{\alpha} q' \wedge p'\mathcal{S}q'$.

q strongly simulates p if there exists a strong simulation $\mathcal{S}$ such that $p\mathcal{S}q$.

Example 2.10. Strong simulation. [1, p. 18, example 3.4]. Consider the two LTS's.



If we define $\mathcal{S}$ to be

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

then $\mathcal{S}$ is a strong simulation. Hence, p_0 strongly simulates q_0, or in other words, q_0 is strongly simulated by p_0, i.e., $q\mathcal{S}p$. This is verified by examining each pair $(q, p) \in \mathcal{S}$ and considering every transition $q \xrightarrow{\alpha} q'$, and show that q is matched by some transition $p \xrightarrow{\alpha} p'$ of p.

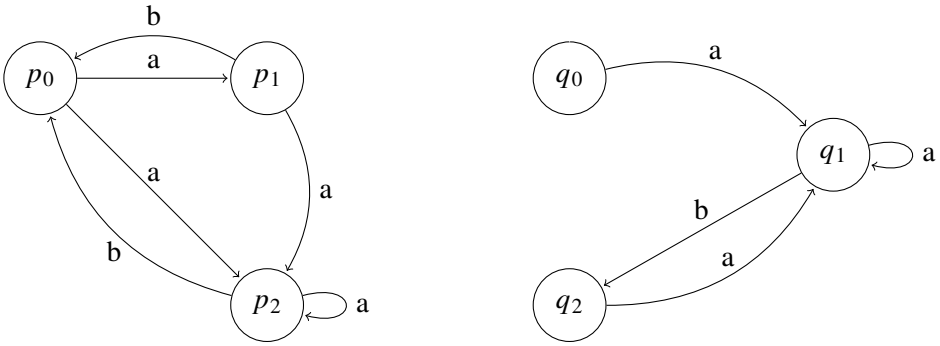
For example, consider the pair $(q_1', p_1) \in \mathcal{S}$. q_1' contains one transition $q_1' \xrightarrow{c} q_3$, which is matched by $p_1 \xrightarrow{c} p_3$ because $(q_3, p_3) \in \mathcal{S}$.

2.3.2 Strong bisimulation

The converse $\mathcal{R}^{-1}$ of any binary relation $\mathcal{R}$ is the set of pairs (y, x) such that $(x, y) \in \mathcal{R}$. A strong bisimulation occurs when both p strongly simulates q and q strongly simulates p.

Definition 2.4. Strong bisimulation, strong equivalence. [1, p. 18, definition 3.6]. A binary relation $\mathcal{S}$ over $\mathcal{Q}$ is a strong bisimulation over the LTS $(\mathcal{Q}, \mathcal{T})$ if both $\mathcal{S}$ and its converse are simulations. If there exists a strong bisimulation $\mathcal{S}$ such that $p\mathcal{S}q$, we say that p and q are strongly bisimilar or strongly equivalent, written $p \sim q$.

Example 2.11. Strong bisimulation. [1, p. 18, example 3.7]. Consider the two LTS's.



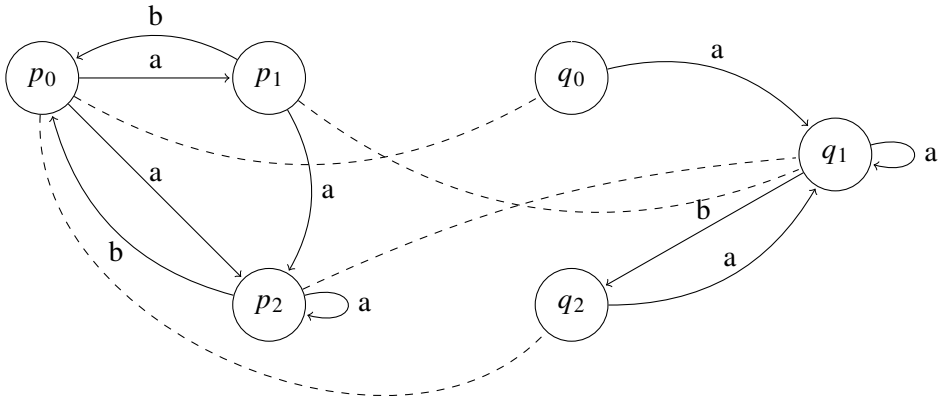
We show that $\mathcal{S}$ is a bisimulation by defining

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

This proves that $p_0 \sim q_0$, i.e., $p_0 \mathcal{S} q_0$.

For example, consider the pair $(p_0, q_0) \in \mathcal{S}$. p_0 contains two transitions $p_0 \xrightarrow{a} p_1$ and $p_0 \xrightarrow{a} p_2$. $p_0 \xrightarrow{a} p_1$ is matched by $q_0 \xrightarrow{a} q_1$ because $(p_1, q_1) \in \mathcal{S}$. This also applies conversely, $q_0 \xrightarrow{a} q_1$ is matched by $p_0 \xrightarrow{a} p_1$ because $(q_1, p_1) \in \mathcal{S}^{-1}$.

It often aids understanding to show a graphical representation of bisimulation by linking the related states on the transition graph, as shown below.



2.3.3 Weak bisimulation

So far, we have explored the rigid notion of equivalence related to strong simulation and strong bisimulation. However, there exist LTS's which have different internal behaviours but may nevertheless be considered equivalent. One such notion of equivalence which addresses this is weak bisimulation.

Any action $\xrightarrow{\lambda}$, with $\lambda \in \mathcal{L}$ is called an observation. An experiment e is a sequence $e = \lambda_1 \ldots \lambda_n$ of observable actions. Internal actions may occur from these observable actions, which are in itself unobservable. For example, the interaction between $\xrightarrow{\bar{a}}$ (output) and $\xrightarrow{a}$ (input).

Definition 2.5. Experiment relations. [1, p. 52, definition 6.1]. The relations $\Rightarrow$ and $\overset{s}{\Rightarrow}$, $\forall s \in \mathcal{L}^*$, are defined as follows:

(1) $P \Rightarrow Q$. There is a sequence of zero or more reactions $P \rightarrow \cdots \rightarrow Q$, i.e., $\Rightarrow \overset{\text{def}}{=} \rightarrow^*$.

(2) $P \overset{s}{\Rightarrow} Q$. Let $s = \alpha_1 \ldots \alpha_n$. There is a sequence of reactions defined by s denoting $P \Rightarrow \xrightarrow{\alpha_1} P_1 \ldots \Rightarrow \xrightarrow{\alpha_n} P_n \Rightarrow Q$, i.e., $\overset{s}{\Rightarrow} \overset{\text{def}}{=} \Rightarrow \xrightarrow{\alpha_1} \Rightarrow \ldots \Rightarrow \xrightarrow{\alpha_n} \Rightarrow$.

$P \Rightarrow Q$ defines an unspecified number of reactions, or transitions, to reach state Q from state P. $P \overset{s}{\Rightarrow} Q$ defines a specified sequence s of transitions to reach state Q from state P.

Definition 2.6. Weak bisimulation. [1, p. 53, definition 6.2, definition 6.5]. Let $\mathcal{S}$ be a binary relation over $\mathcal{P}$. Then $\mathcal{S}$ is a weak bisimulation if, whenever $P\mathcal{S}Q$,

(1) if $P \overset{e}{\Rightarrow} P'$, then $\exists Q' \in \mathcal{P} : Q \overset{e}{\Rightarrow} Q' \wedge P'\mathcal{S}Q'$, and

(2) the converse of (1) on the actions from $\mathcal{Q}$.

P and Q are weakly bisimiliar, weakly equivalent, or observation equivalent, written $P \approx Q$, if there exists a weak bisimulation $\mathcal{S}$ such that $P\mathcal{S}Q$.

Proposition 2.1. [1, p. 54, proposition 6.6]. A strong bisimulation implies a weak bisimulation, i.e., $P \sim Q$ implies $P \approx Q$.

Example 2.12. Weak bisimulation. [17, p. 111, example 4.2.3]. Consider the following weak bisimulation

$$\tau.a \approx a$$

To prove this is a weak bisimulation $\mathcal{S}$, we define $\mathcal{S}$ to be

$$\mathcal{S} = \{(a, a), (0, 0), (\tau.a, a)\}$$

The process $\tau.a$ has two possible transitions $\tau.a \overset{\tau}{\Rightarrow} a$ and $\tau.a \overset{a}{\Rightarrow} 0$. The first is matched by $a \Rightarrow a$, i.e., $a\mathcal{S}a$, and the second is matched by $a \overset{a}{\Rightarrow} 0$, i.e., $0\mathcal{S}0$. Conversely, the process a has one possible transition $a \overset{a}{\Rightarrow} 0$. This is matched by $\tau.a \overset{a}{\Rightarrow} 0$, i.e., $\tau.a\mathcal{S}a$. This move matching is often referred to as the bisimulation game.

The above are the fundamentals of bisimulation. Many notions of bisimulation exist, which include environmental bisimulations [18] and up-to bisimulations [19]. We will later encounter a notion of bisimulation specific to the π-calculus in fresh-register automata.

2.4 Fresh-register automata

In the π-calculus examples, we explored how placing processes in any context can lead to a potentially infinite number of states. This is due to the infinite set of names $\mathcal{N}$ and names arising from dynamic name generation. As a result, representing systems in an LTS can be a challenge. By using a new class of automata developed by Tzevelekos [5] called fresh-register automata (FRA), we are presented with an approach which allows for many infinite-state LTS's to be represented finitely.

We begin by exploring the rationale behind FRA's. The creation of new entities is a common abstraction in programming. For example, the declaration of a new object in *Java*, the allocation of memory in *C*, or the declaration of a restricted name νa in the

π-calculus. These entities can be created at will and in such a manner that newly created entities are always *fresh*, i.e., it has never been generated before. We call these entities *names* [20]. As a result, the set of names is infinite. Finite-memory automata (FMA) operate over infinite alphabets, introduced by Kaminski and Francez [21] in 1994. An FMA consists of an automaton attached with a finite number of name-storing registers.

An FMA can access its registers by either comparing an input name to a stored one, or by storing an input name in one of its registers. A name is stored only if it is *locally fresh*, i.e., it does not appear in any register. Therefore, FMA's are history-free, meaning that their computational steps rely only on their current registers.

Example 2.13. Local freshness. In relation to the π-calculus, consider the process

$$a(x).P$$

In an FMA, when we encounter the input name x, its name originates from the context. It could be either a name which we encountered before (like a, b, etc.) or an entirely new name, i.e., a locally fresh name.

FRA's extend FMA's by capturing *global freshness*, giving rise to a basic automata-theoretic model of names. An input name is stored in a register just in case it is fresh in the whole current run. This history-sensitive feature precisely captures fresh-name creation.

Example 2.14. Global freshness. Consider the π-calculus expression given by Tzevelekos [5, p. 303, example 31]

$$\nu b.p(a,b) \quad \textit{with definition} \quad p(a,b) = \bar{a}\langle b\rangle.\nu c.p(b,c)$$

We repeatedly generate a fresh name and output this name. A fresh name is created by the restriction νb (and νc). When the output action is performed, we send this name over to a receiver in the context. The environment can potentially remember this fresh name (i.e., its name is globally fresh), along with the subsequent infinite stream of fresh names. So the FRA accepts this fresh name (which we did not see before), stores it in a register, and repeats this process.

In fact, FMA's cannot represent the above expression due to fresh-name creation. However, FRA's are able to represent this using a single state with one register, which we will see in a later chapter.

We build up the definition of FRA. There are two sets of input symbols:

- $\mathbb{A}$ is an infinite set of *names*. Let characters $a, b,$ etc. range over names.
- $\mathbb{C}$ is a finite set of *constants*. Constants have an auxiliary role and are non-storable.

$\mathbb{A}^*$ is the set of finite strings of names.
$\mathbb{A}^{\circledast}$ is the set of finite strings of names containing only pairwise distinct names.
$\vec{a} = a_1 \ldots a_n$ is a vector of strings.
$\mathsf{img}(\vec{a}) = \{a_1, ..., a_n\}$ is a set of strings.

For each $n \in \omega$ (ordinal number), $[n]$ is the set $\{1, ..., n\}$. We define

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

where labels of the form i are known transitions.
$i^\bullet$ is a locally fresh transition.
$i^{\circledast}$ is a globally fresh transition.

$\mathbb{L}_n$ is the set of labels generated by $[n]$. $\mathbb{L}_n$ consists of the set of constants $\mathbb{C}$ combined with transition labels from the set $[n]$. Let

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

be the set of *register assignments* of size n. The register assignment σ is a function which maps elements from the set $[n]$ to a name in the set of names $\mathbb{A}$ combined with the special name $\sharp$. The only name which can exist more than once in the registers is $\sharp$. Let

$$\mathsf{img}(\sigma) = \{\, a \in \mathbb{A} \mid \exists i.\ \sigma(i) = a \,\}$$

be the name-range of σ, i.e., the set of all the names which exist in the register assignment. Let

$$\mathsf{dom}(\sigma) = \{\, i \in [n] \mid \sigma(i) \in \mathbb{A} \,\}$$

be the domain of σ, i.e., the set of all transition labels which map to a name in the register assignment. Whenever $a \notin \mathsf{img}(\sigma)$, i.e., a name is not in the register assignment,

$$\sigma[i \mapsto a] = \{\, (i, a) \,\} \cup \{\, (j, \sigma(j)) \mid j \in [n] \setminus \{i\} \,\}$$

is an update of σ, for any $i \in [n]$. This update operation adds the label-name pair (i, a) to the registers, or overwrites the label-name pair if the label already exists in a register.

Definition 2.7. Fresh-register automaton (FRA). [5, p. 296, definition 1]. An FRA of n registers is a quintuple $\mathcal{A} = \langle Q, q_0, \sigma_0, \delta, F \rangle$ where:

- Q is a finite set of states,
- q_0 is the initial state,
- $\sigma_0 \in \mathsf{Reg}_n$ 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.

$\mathcal{A}$ is a **register automaton (RA)** if there are no globally fresh transitions, i.e., $\nexists q, q', i$ such that $(q, i^{\circledast}, q') \in \sigma$.

Example 2.15. Fresh-register automata. [5, p. 296]. Suppose $\mathcal{A}$ is at state q_1 with the current register assignment σ. If input $\ell \in \mathbb{C} \cup \mathbb{A}$ arrives then:

- If $\ell \in \mathbb{C}$ and $(q_1, \ell, q_2) \in \delta$ then $\mathcal{A}$ accepts ℓ and moves to q_2. This means that ℓ is a label constant and the transition ℓ is taken to reach state q_2.

- If $\ell \in \mathbb{A}$ and $(q_1, i, q_2) \in \delta$ and $\sigma(i) = \ell$ then $\mathcal{A}$ accepts ℓ and moves to q_2. In this case, ℓ is a name, so a lookup is performed on the registers σ to find the label i. This transition i is taken to reach state q_2.
- If $\ell \in \mathbb{A}$ and $(q_1, i^\bullet, q_2) \in \delta$ and ℓ is not stored in σ then $\mathcal{A}$ accepts ℓ, it sets $\sigma(i) = \ell$ and moves to q_2. This is a locally fresh transition. Since the name ℓ does not exist in the registers σ, an update is performed and a label i in σ is set to ℓ. The transition i is taken to reach state q_2.
- If $\ell \in \mathbb{A}$ and $(q_1, i^\circledast, q_2) \in \delta$ and $\ell \notin \mathsf{img}(\sigma_0)$ and ℓ has not appeared in the current run then $\mathcal{A}$ accepts ℓ, it sets $\sigma(i) = \ell$ and moves to q_2. This is a globally fresh transition. In this case, the name ℓ has never appeared in the registers σ. As a result, an update is performed and a label i in σ is set to ℓ and the transition i is taken to reach state q_2.

The above example is formally defined by means of *configurations*. Configurations represent the intended current state of the automaton. In addition to the state, the configuration contains information on the current register assignment and the *history*. The history is the set of names that have appeared thus far, which is a component necessary for globally fresh transitions.

Definition 2.8. Configuration. [5, p. 297, definition 2]. A configuration of $\mathcal{A}$ is a triple $(q, \sigma, H) \in \hat{Q}$, where q is the state, σ is the register assignment and H represents the history, with

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

and $\mathcal{P}_{\mathsf{fn}}(\mathbb{A})$ being the set of finite subsets of $\mathbb{A}$. From δ define a transition on configurations

$$\longrightarrow_\delta \subseteq \hat{Q} \times (\mathbb{C} \cup \mathbb{A}) \times \hat{Q}$$

as follows. For all $(q, \sigma, H) \in \hat{Q}$ and $(q, \ell, q') \in \delta$:

- If $\ell \in \mathbb{C}$ then $(q, \sigma, H) \xrightarrow{\ell}_\delta (q', \sigma, H)$. The transition ℓ is taken from the configuration (q, σ, H) to reach the configuration (q', σ, H).
- If $\ell = i$ and $\sigma(i) = a$ then $(q, \sigma, H) \xrightarrow{a}_\delta (q', \sigma, H \cup \{a\})$. ℓ is a label that corresponds with the name a in the registers σ. The transition a is taken to reach the configuration $(q', \sigma, H \cup \{a\})$, where a is added to the history.
- If $\ell = i^\bullet$ and $a \notin \mathsf{img}(\sigma)$ then $(q, \sigma, H) \xrightarrow{a}_\delta (q', \sigma', H')$ with $\sigma' = \sigma[i \mapsto a]$ and $H' = H \cup \{a\}$. This is a locally fresh transition. If the name a does not exist in the registers σ, a transition is taken to the configuration (q', σ', H'). In this configuration, the registers are updated to include the label-name pair (i, a) and the history adds a.
- If $\ell = i^\circledast$ and $a \notin H \cup \mathsf{img}(\sigma_0)$ then $(q, \sigma, H) \xrightarrow{a}_\delta (q', \sigma', H')$ with $\sigma' = \sigma[i \mapsto a]$ and $H' = H \cup \{a\}$. This is a globally fresh transition. If the name a does not exist in either the history or the registers, a transition is taken to the configuration (q', σ', H'). Like with the locally fresh transition, the registers are updated with the name a and added to the history.

Language 3

3.1 The extended pi-calculus

To directly represent the π-calculus by fresh-register automata, Tzevelekos defines the extended π-calculus, or $\times\pi$-calculus.

Definition 3.1. **The $\times\pi$-calculus.** [5, p. 302, definition 29]. The set of main constructions Π is defined by the following:

$P, Q ::=$	process
$a(b).P$	input
$\bar{a}\langle b\rangle.P$	output
$[a{=}b]P$	equality
$[a{\neq}b]P$	inequality†
$\nu a.P$	restriction
$P + Q$	summation
$P \mid Q$	composition
$p(\vec{a})$	process
0	inaction

The sets of intermediate constructions Π_{out} and Π_{inp} are defined as:

$$P_{\mathsf{out}} ::= b.P \;\mid\; \nu a.P_{\mathsf{out}} \;\mid\; P \mid P_{\mathsf{out}} \;\mid\; P_{\mathsf{out}} \mid P$$
$$P_{\mathsf{inp}} ::= (b).P \;\mid\; \nu a.P_{\mathsf{inp}} \;\mid\; P \mid P_{\mathsf{inp}} \;\mid\; P_{\mathsf{inp}} \mid P$$

We write $\hat{\Pi}$ for $\Pi \cup \Pi_{\mathsf{out}} \cup \Pi_{\mathsf{inp}}$ and let $\hat{P}, \hat{Q}, \ldots$ range over its elements.

The purpose of the intermediate constructions Π_{out} and Π_{inp} is to account for the fact that transitions are multi-symbol and the FRA recognises one symbol at a time [5, p. 302]. The main constructions Π are near-identical to the original π-calculus, except for some small additions and changes.

- The equality construct is introduced. This means P is run only if the two names a and b are equal, equivalent to an 'if' statement.
- † Although not defined by Tzevelekos, we introduce the inequality term. This is the negated equality condition.
- Replication $P!$ is replaced by a process definition $p(\vec{a}) = P$. This means that processes can be called with (or without) a vector of names as arguments. This is similar to a function call in programming languages.
- A cosmetic change of a dot is placed between a and P in restriction.

3.2 Lexer

The lexer is implemented in *Ragel* [22], which generates a lexical analyser in *Go* code.

The $\times\pi$-calculus syntax is converted into ASCII characters for standard input and parsing. The following are the ASCII counterparts.

$\times\pi$-calculus syntax	*ASCII form*	
$P ::=$	`P =`	process
$a(b).P$	`a(b).P`	input
$\bar{a}\langle b\rangle.P$	`a'<b>.P`	output
$[a{=}a]P$	`[a=a]P`	equality
$[a{\neq}b]P$	`[a!=b]P`	inequality
$\nu a.P$	`$a.P`	restriction
$P + Q$	`P + Q`	summation
$P \mid Q$	`P | Q`	composition
$P(a,b,c)$	`P(a,b,c)`	process
0	`0`	inaction

Tab. 3.1: $\times\pi$-calculus syntax and their ASCII counterparts.

The special characters (`=`, `$`, `+`, etc.) are recognised as single characters while the names `a`, `b`, `c`, `P` and `Q` are expressed as a regular expression `[a-zA-z0-9]+`.

3.3 Grammar

The parser is implemented with *goyacc* [23], a version of *Yacc* (Yet Another Compiler-Compiler) in *Go* for generating the parser. The parser generated is LR(1), which means that the input is parsed bottom-up and that there is a lookahead of one token.

The grammar of the language is described in Backus-Naur form (BNF), with the exception of ⟨*ident*⟩, which is represented as a regular expression.

⟨*ident*⟩ ::= [a-zA-z0-9]+

⟨*stmts*⟩ ::= ‘’
| ⟨*stmts*⟩ ⟨*stmt*⟩

⟨*stmt*⟩ ::= ⟨*decl-process*⟩
| ⟨*decl-params-process*⟩
| ⟨*undecl-process*⟩

⟨*decl-process*⟩ ::= ⟨*ident*⟩ ‘=’ ⟨*elem*⟩

⟨*decl-params-process*⟩ ::= ⟨*ident*⟩ ‘(’ ⟨*decl-params*⟩ ‘=’ ⟨*elem*⟩

⟨*decl-params*⟩ ::= ⟨*ident*⟩ ‘,’ ⟨*decl-params*⟩
| ⟨*ident*⟩ ‘)’

⟨*undecl-process*⟩ ::= ⟨*elem*⟩

⟨*elem*⟩ ::= ⟨*parentheses*⟩
| ⟨*parallel*⟩
| ⟨*sum*⟩
| ⟨*output*⟩
| ⟨*input*⟩
| ⟨*equality*⟩
| ⟨*inequality*⟩
| ⟨*restriction*⟩
| ⟨*nil*⟩
| ⟨*process*⟩
| ⟨*params-process*⟩

⟨*parentheses*⟩ ::= ‘(’ ⟨*elem*⟩ ‘)’

⟨*parallel*⟩ ::= ⟨*elem*⟩ ‘|’ ⟨*elem*⟩

⟨*sum*⟩ ::= ⟨*elem*⟩ ‘+’ ⟨*elem*⟩

⟨*output*⟩ ::= ⟨*ident*⟩ ‘'’ ‘<’ ⟨*ident*⟩ ‘>’ ‘.’ ⟨*elem*⟩
| ⟨*ident*⟩ ‘<’ ⟨*ident*⟩ ‘>’ ‘.’ ⟨*elem*⟩

⟨*input*⟩ ::= ⟨*ident*⟩ ‘(’ ⟨*ident*⟩ ‘)’ ‘.’ ⟨*elem*⟩

⟨*equality*⟩ ::= ‘[’ ⟨*ident*⟩ ‘=’ ⟨*ident*⟩ ‘]’ ⟨*elem*⟩

⟨*inequality*⟩ ::= ‘[’ ⟨*ident*⟩ ‘!=’ ⟨*ident*⟩ ‘]’ ⟨*elem*⟩

⟨*restriction*⟩ ::= ‘$’ ⟨*name*⟩ ‘.’ ⟨*elem*⟩

⟨*nil*⟩ ::= ‘0’

⟨*process*⟩ ::= ⟨*ident*⟩

⟨*params-process*⟩ ::= ⟨*ident*⟩ ‘(’ ⟨*params*⟩

⟨*params*⟩ ::= ⟨*ident*⟩ ‘,’
| ⟨*name*⟩ ‘)’

3.3.1 Shift-reduce conflicts

The main challenge in constructing the grammar was the resolution of shift-reduce conflicts. Shift-reduce conflicts result from ambiguous grammars.

Operator associativity

There are two binary operators in the $\times\pi$-calculus, summation $P + Q$ and composition $P \mid Q$. Without specifying associativity and given the expression $P + Q \mid R$, this can be interpreted as $(P + Q) \mid R$ or $P + (Q \mid R)$, resulting in ambiguity.

Operator precedence is usually reserved for practical use in process calculi, as in this case, so theoretical papers and books usually do not discuss it. Neither Milner nor Tzevelekos specifies a precedence. As a result, we define that summation binds more tightly than composition. We specify that + has higher precedence and both + and | are right associative. In *Yacc*, we declare

```
%right VERTBAR
%right PLUS
```

So $A + B + C \mid D \mid E \mid F$ is parsed as $(A + (B + C)) \mid (D \mid (E \mid F))$.

Grammar rules precedence

Specifying precedence in grammar rules is also required to resolve conflicts. For example, we encounter a conflict if we do not provide precedence, written `%prec TOKEN` in *Yacc*, in the following grammar.

⟨*process*⟩ ::= ⟨*ident*⟩ `%prec LOWER`
| ⟨*ident*⟩ '`(`' ⟨*params*⟩

This specifies that the first statement has a lower precedence than the left bracket of the second statement. This informs the parser that when we encounter ⟨*indent*⟩ and a left bracket proceeds, then the rule must be the second statement.

3.4 Elements

The $\times\pi$-calculus constructs consist of structs of type `Element`, given below.

```
type Element interface {
    Type() ElementType
}
```

Listing 3.1: The element base type of the $\times\pi$-calculus constructs.

The `Element` type is subtyped into its specific construct. These subtypes contain relevant information about that construct. All elements implement the `Type()` function to allow

for identification. A data type representing a name is common to most constructs and is given below.

```
type Name struct {
    Name string
    Type NameType
}
```

Listing 3.2: The name struct.

Channels and variables are represented by the struct `Name`. The type of the name is either free or bound. We now describe the elements from the main constructions set Π.

Nil

```
type ElemNil struct {}
```

Listing 3.3: The nil element struct.

The simplest element is the nil process. The struct holds no data and serves as a terminal node.

Process

```
type ElemProcess struct {
    Name       string
    Parameters []Name
}
```

Listing 3.4: The process element struct.

The only other terminal node is the process. The process represents both the process P and parameterised process $p(\vec{a})$. In the process P without parameters, the `Name` is P and the `Parameters` are empty. In $p(a, b, c)$, the `Parameters` are a list of names a, b and c.

Output

```
type ElemOutput struct {
    Channel Name
    Output  Name
    Next    Element
}
```

Listing 3.5: The output element struct.

In $\bar{a}\langle b\rangle.P$, the element `ElemOutput` holds `Channel` a and `Output` b. The next element is an `ElemProcess` P.

Input

```
type ElemInput struct {
    Channel Name
    Input   Name
    Next    Element
}
```

Listing 3.6: The input element struct.

In $a(x).a\langle b\rangle.P$, `ElemInput` holds `Channel` a and `Input` x. The next element is an `ElemOutput`, followed by an `ElemProcess` P.

Equality and inequality

```
type ElemEquality struct {
    Inequality bool
    NameL      Name
    NameR      Name
    Next       Element
}
```

Listing 3.7: The equality element struct.

In $[a=b]P$, `ElemEquality` holds `NameL` a and `NameR` b. The next element is an `ElemProcess` P. In $[a\neq b]P$, the element holds the same data, with the addition that `Inequality` is true.

Restriction

```
type ElemRestriction struct {
    Restrict Name
    Next     Element
}
```

Listing 3.8: The restriction element struct.

In $\nu a.P$, `ElemRestriction` holds a as `Restrict`. The next element is an `ElemProcess` P.

Sum

```
type ElemSum struct {
    ProcessL Element
    ProcessR Element
}
```

Listing 3.9: The sum element struct.

In $a(x).0+b\langle c\rangle.0$, `ElemSum` contains `ElemInput` as `ProcessL` and `ElemOutput` as `ProcessR`.

Parallel

```
type ElemParallel struct {
    ProcessL Element
    ProcessR Element
}
```

Listing 3.10: The parallel element struct.

Replacing the previous sum expression with a parallel, $a(x).0 \mid b\langle c\rangle.0$, we get an `ElemParallel` containing `ElemInput` as `ProcessL` and `ElemOutput` as `ProcessR`.

3.5 Abstract syntax tree

Together with the grammar and element definitions, the abstract syntax tree (AST) of the processes can be built. Action symbols are added to the grammar to construct a symbol table containing information on how to structure the AST's. The procedure is best described using examples with increasing complexity.

Example 3.1. **Unary elements.** We define unary elements to be output, input, restriction, and equality/inequality because they point to only one element. We consider the nil and process constructs to be nullary elements because they are terminal elements, i.e., they do not point to any element. Consider the $\times\pi$-calculus expression in ASCII form

```
$a.b'<a>.c(x).[b=x]0
```

The expression contains unary $\times\pi$-calculus constructs in the order restriction, output, input, and equality, and ends with the nullary element nil.

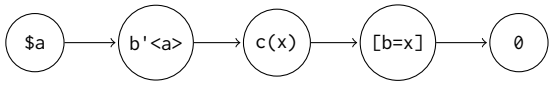


Fig. 3.1: Unary elements restriction, output, input, and equality in an AST.

The first element that the bottom-up parser encounters is nil, then equality, then the elements up to restriction. The symbol table contains a variable containing the current element. This variable is assigned as the terminal element `ElemNil` when encountered. When it encounters the subsequent construct, it points this new element to the current element and assigns itself as the current element. As a result, each element is linked together by using the `Next` struct member, creating a linked list.

Example 3.2. **Binary elements.** In contrast to the unary elements above, we call summation and composition binary elements due to their ability to point to two processes. Consider

```
a(x).P | b'<c>.0 | Q
```

The order in which the elements are parsed are `P`, `a(x)`, `|`, `0`, `b'<c>`, `|`, and `Q`. When the first `|` is reached, the current element comprises of `a(x).P`. The current element is

pushed to a stack and the current element variable is reset. When the second `|` is reached, the current element `b'<c>.0` is again pushed to the stack and the variable is reset. The final element is parsed, which is `Q`. At this point, all elements (for this example) are popped off the stack to form a right-leaning `ElemParallel` chain.

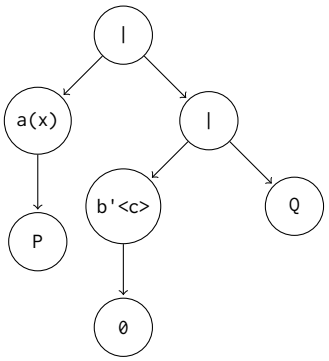


Fig. 3.2: The binary element parallel in an AST comprises of two child elements.

The same algorithm is used for summation, with the use of a different stack.

Example 3.3. Parenthesised processes. The use of parentheses introduces the concept of scope to the language. Consider

```
a(b).(P + ((A + B) | C) + R)
```

where we add nested parentheses to specify bindings. Processes are enclosed inside brackets. We need to keep track of how many summations and compositions that occur inside the current bracket level so we know how many elements to pop off their stacks.

The elements parsed in order are `(`, `P`, `+`, `(`, `(`, `A`, `+`, `B`, `)`, `|`, `C`, `)`, `+`, `R`, `)`, and `a(b)`. When an opening parenthesis is encountered, the number of summations are saved to a stack and the number of summations are reset to zero. When a '+' is parsed, this value is incremented. When the final right-hand side of the summation process in the current bracket level is parsed (in the case of `(A + B)`, this is `B`), elements are popped from the stack up to the current number of summations. Ultimately, a closing parenthesis is encountered and the number of summations from the previous scope is popped from the stack and restored. The same procedure follows for composition.

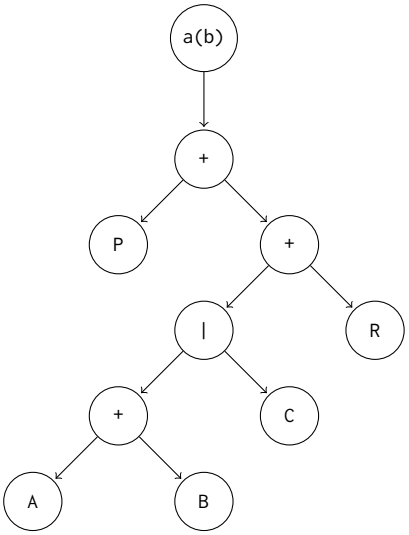


Fig. 3.3: Resultant AST of a parenthesised expression which captures the scoping of elements.

Example 3.4. Declared processes. So far, we have only described a single undeclared process, i.e., processes without the prefix '`P =`'. However, the grammar supports multi-line statements of declared processes with names and parameters. An undeclared process initialises the program, similar to a main function. It is expected that only one undeclared process exists. Consider the program

```
P(a,b) = a(x).b'<b>.0
Q = c(y).0
P(i,j) | Q
```

where we declare `P` with parameters `a` and `b`, `Q` with no parameters and an undeclared process. From this program, we gather three AST's.

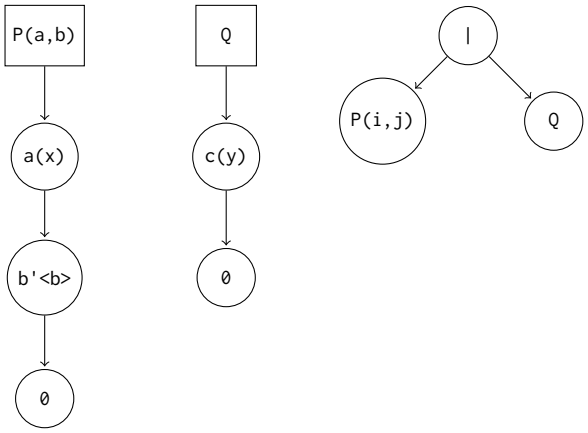


Fig. 3.4: Declared processes `P(a,b)` and `Q`, along with an undeclared parallel process.

Declared processes are stored in a map which maps the process name to a struct containing the parameters and associated process. In this case, `P` is mapped to a struct containing `a` and `b` as the parameters and `a(x).b'<b>.0` as the process. `Q`, with no pa-

rameters, is mapped to `c(y).0`. In the diagram, these are represented by rectangles. The sole undeclared process is stored as a variable.

Root element

In anticipation of the implementation of the $\times\pi$-calculus transition relation, a root element is appended to the top-most undeclared element. This allows for recursive AST manipulation operations to be implemented cleanly, e.g., if we want to replace the top-most element, we simply point the root's `Next` to the new element instead of keeping track a head variable. The root acts as the head of the element and is never replaced.

3.6 Alpha-conversion

The name b is bound in restriction $\nu b.P$ and input $a(b).P$. We denote the bound names of P to be $\mathsf{bn}(P)$, and the free names, i.e., not bound, of P to be $\mathsf{fn}(P)$. The changing of a bound name into a fresh name, i.e., a name not encountered before, is called *alpha-conversion* [1, p. 29]. Alpha-conversion is performed in order to accomplish substitution properly.

Example 3.5. Alpha-conversion. Consider the expression

$$a(a).\nu a.a(a).a\langle a\rangle.0$$

It is difficult to differentiate between the free names and the scope of the bound names. When we alpha-convert, we obtain

$$a(a_1).\nu a_2.a_2(a_3).a_3\langle a_3\rangle.0$$

where the subscript numbered a's are different bound names. Note how the restriction name a_2 overshadows a_1, and how the input name a_3 does the same for a_2.

There are approaches to resolving alpha-conversion practically. One such notion is De Bruijn indices which originates from the lambda calculus [24]. This does so without naming the bound variables and assigns indices to the binders. However, we choose a simpler approach that converts the expression to a form which follows the Barendrecht-convention. In the expression, all bound names are uniquely named corresponding to their binders, as with the example above.

An algorithm was written to perform this, and its pseudocode is given below. The element AST is traversed. When an input or restriction is encountered, a bound name is generated in the form `"x_1"`, `"y_2"`, etc., and takes the place of the original name. The tree is traversed again from that point, and this time the original name is replaced by the new bound name, up until an input or restriction with the same original name.

Algorithm 3.1 Performs alpha-conversion on a process

```
1:  function ALPHACONVERT(elem)
2:      switch elem.TYPE() do
3:          case INPUT
4:              oldName ← elem.Input.Name
5:              boundName ← GENERATEBOUNDNAME()
6:              elem.Input ← NAME{boundName, BOUND}
7:              SUBSTITUTEBOUNDNAMES(elem.Next, oldName, boundName)
8:              ALPHACONVERT(elem.Next)
9:          case RESTRICTION
10:             oldName ← elem.Restrict.Name
11:             boundName ← GENERATEBOUNDNAME()
12:             elem.Restrict ← NAME{boundName, BOUND}
13:             SUBSTITUTEBOUNDNAMES(elem.Next, oldName, boundName)
14:             ALPHACONVERT(elem.Next)
15:         case UNARY
16:             ALPHACONVERT(elem.Next)
17:         case BINARY
18:             ALPHACONVERT(elem.ProcessL)
19:             ALPHACONVERT(elem.ProcessR)
20: function SUBSTITUTEBOUNDNAMES(elem, oldName, boundName)
21:     switch elem.TYPE() do
22:         case INPUT
23:             if elem.Channel.Name = oldName then
24:                 elem.Channel ← NAME{boundName, BOUND}
25:             if elem.Input.Name ≠ oldName then
26:                 SUBSTITUTEBOUNDNAMES(elem.Next, oldName, boundName)
27:         case RESTRICTION
28:             if elem.Restrict.Name ≠ oldName then
29:                 SUBSTITUTEBOUNDNAMES(elem.Next, oldName, boundName)
30:         case PROCESS
31:             for i, param ∈ elem.Parameters do
32:                 if param.Name = oldName then
33:                     elem.Parameters[i] ← NAME{boundName, BOUND}
34:         case UNARY
35:             for data ∈ {Channel, Output, NameL, NameR} do
36:                 if elem.data.Name = oldName then
37:                     elem.data ← NAME{boundName, BOUND}
38:             SUBSTITUTEBOUNDNAMES(elem.Next, oldName, boundName)
39:         case BINARY
40:             SUBSTITUTEBOUNDNAMES(elem.ProcessL)
41:             SUBSTITUTEBOUNDNAMES(elem.ProcessR)
```

Transition Relation

4

The transition relation, or transition rules, describes instructions to translate the $\times\pi$-calculus to fresh-register automata. We formalise the LTS of the $\times\pi$-calculus with the following definitions.

Definition 4.1. States. [5, p. 303]. The set of states is defined as $O(\hat{K})$ with each a process-in-context $O(\sigma, \hat{P})$, denoted as $\sigma \vdash \hat{P}$.

Each state is also called a configuration which borrows the concept from FRA's. Only states from $O(K)$ are final, where there are no intermediate stages in the process P.

Definition 4.2. Labels. [5, p. 303, definition 30]. The set of labels α is defined as

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

where $i, j \in \omega$.

The transition labels borrow notation from FRA's, with $i^{\bullet}$ meaning fresh inputs and $i^{\circledast}$ representing fresh outputs.

The full transition relation defined by Tzevelekos is given in table 4.1. The rules follow structural operational semantics which is in the form of inference rules. We refer to the transition(s) at the top the line as the premise(s) and the transition at the bottom of the line as the conclusion. As a general guide to interpreting the rules, if the starting state $O(\hat{K})$ of the conclusion is pattern matched, then try performing the transition(s) denoted in premise(s). If a transition can be taken, then the next state $O(\hat{K})$ of the conclusion can be formed. Side conditions to the right of the rule may exist, which must be fulfilled in order to take the transition. We will explore each rule in greater depth in the later section.

$$\textsc{Inp1}\quad \frac{}{\sigma \vdash a(b).P \xrightarrow{i} \sigma \vdash (b).P}\ \sigma(i)=a$$

$$\textsc{Inp2a}\quad \frac{}{\sigma \vdash (b).P \xrightarrow{i} \sigma \vdash P\{a/b\}}\ \sigma(i)=a$$

$$\textsc{Inp2b}\quad \frac{}{\sigma \vdash (b).P \xrightarrow{i^\bullet} \sigma[i \mapsto b] \vdash P}\ \begin{array}{l} i=\min\{\, i \mid \sigma(i) \notin \mathsf{fn}(P)\,\} \\ b \notin \mathsf{img}(\sigma) \end{array}$$

$$\textsc{DblInp}\quad \frac{\sigma \vdash P \xrightarrow{i} \sigma \vdash P_{\mathsf{inp}} \xrightarrow{j/j^\bullet} \sigma' \vdash P'}{\sigma \vdash P \xrightarrow{ij/ij^\bullet} \sigma' \vdash P'}$$

$$\textsc{Out1}\quad \frac{}{\sigma \vdash \bar{a}b.P \xrightarrow{i} \sigma \vdash b.P}\ \sigma(i)=a \qquad \textsc{Out2}\quad \frac{}{\sigma \vdash b.P \xrightarrow{i} \sigma \vdash P}\ \sigma(i)=b$$

$$\textsc{DblOut}\quad \frac{\sigma \vdash P \xrightarrow{i} \sigma \vdash P_{\mathsf{out}} \xrightarrow{j/j^\circledast} \sigma' \vdash P'}{\sigma \vdash P \xrightarrow{\bar{i}j/\bar{i}j^\circledast} \sigma' \vdash P'}$$

$$\textsc{Res}\quad \frac{(\sigma + a) \vdash \hat{P} \xrightarrow{\alpha} (\sigma' + a) \vdash \hat{P}'}{\sigma \vdash \nu a.\hat{P} \xrightarrow{\alpha} \sigma' \vdash \nu a.\hat{P}'}\ \alpha \neq (|\sigma|)+1$$

$$\textsc{Open}\quad \frac{\sigma[i \mapsto a] \vdash P_{\mathsf{out}} \xrightarrow{i} \sigma[i \mapsto a] \vdash P}{\sigma \vdash \nu a.P_{\mathsf{out}} \xrightarrow{i^\circledast} \sigma[i \mapsto a] \vdash P}\ \begin{array}{l} i=\min\{\, i \mid \sigma(i) \notin \mathsf{fn}(P)\,\} \\ a \notin \mathsf{img}(\sigma) \end{array}$$

$$\textsc{Match}\quad \frac{\sigma \vdash P \xrightarrow{\alpha} \sigma \vdash \hat{P}'}{\sigma \vdash [a=a]P \xrightarrow{\alpha} \sigma \vdash \hat{P}'} \qquad \textsc{Rec}\quad \frac{\sigma \vdash P\{\vec{a}/\vec{b}\} \xrightarrow{\alpha} \sigma \vdash \hat{P}'}{\sigma \vdash p(\vec{a}) \xrightarrow{\alpha} \sigma \vdash \hat{P}'}\ p(\vec{b})=P$$

$$\textsc{Sum}\quad \frac{\sigma \vdash P \xrightarrow{\alpha} \sigma \vdash \hat{P}'}{\sigma \vdash P + Q \xrightarrow{\alpha} \sigma \vdash \hat{P}'} \qquad \textsc{Sum}_{\textsc{sym}}\quad \frac{\sigma \vdash Q \xrightarrow{\alpha} \sigma \vdash \hat{Q}'}{\sigma \vdash P + Q \xrightarrow{\alpha} \sigma \vdash \hat{Q}'}$$

$$\textsc{Par1}\quad \frac{\sigma \vdash P \xrightarrow{\alpha} \sigma \vdash \hat{P}'}{\sigma \vdash P \mid Q \xrightarrow{\alpha} \sigma \vdash \hat{P}' \mid Q} \qquad \textsc{Par1}_{\textsc{sym}}\quad \frac{\sigma \vdash Q \xrightarrow{\alpha} \sigma \vdash \hat{Q}'}{\sigma \vdash P \mid Q \xrightarrow{\alpha} \sigma \vdash P \mid \hat{Q}'}$$

$$\textsc{Par2}\quad \frac{\sigma \vdash \hat{P} \xrightarrow{i^\bullet/i^\circledast} \sigma[i \mapsto b] \vdash P'}{\sigma \vdash \hat{P} \mid Q \xrightarrow{j^\bullet/j^\circledast} \sigma[j \mapsto b] \vdash P' \mid Q}\ j=\min\{\, j \mid \sigma(j) \notin \mathsf{fn}(P', Q)\,\}$$

$$\textsc{Par2}_{\textsc{sym}}\quad \frac{\sigma \vdash \hat{Q} \xrightarrow{i^\bullet/i^\circledast} \sigma[i \mapsto b] \vdash Q'}{\sigma \vdash P \mid \hat{Q} \xrightarrow{j^\bullet/j^\circledast} \sigma[j \mapsto b] \vdash P \mid Q'}\ j=\min\{\, j \mid \sigma(j) \notin \mathsf{fn}(P, Q')\,\}$$

$$\textsc{Comm}\quad \frac{\sigma \vdash P \xrightarrow{\bar{i}j} \sigma \vdash P' \qquad \sigma \vdash Q \xrightarrow{ij} \sigma \vdash Q'}{\sigma \vdash P \mid Q \xrightarrow{\tau} \sigma \vdash P' \mid Q'}$$

$$\textsc{Comm}_{\textsc{sym}}\quad \frac{\sigma \vdash P \xrightarrow{ij} \sigma \vdash P' \qquad \sigma \vdash Q \xrightarrow{\bar{i}j} \sigma \vdash Q'}{\sigma \vdash P \mid Q \xrightarrow{\tau} \sigma \vdash P' \mid Q'}$$

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

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

Tab. 4.1: The transition relation for the $\times\pi$-calculus [5, p. 304, table 1].

4.1 Ambiguous transition rules

If we study closely the transition rules, there are alternative levels of the transition derivation tree where the rules can be applied. In other words, there are multiple ways of generating the same LTS. Take the configuration

$$\{(1, a)\} \vdash [a{=}a]a(b).0$$

The rule MATCH is pattern matched, and so we try the transition of the premise with $a(b).0$. INP1 is pattern matched, which becomes $(b).0$ after taking the transition with label 1. The MATCH premise states that any label α, either single or double, can be taken which leads to any process $\hat{P}$, intermediate or not, as in this case. As a result, we can take the MATCH conclusion.

However, from this point $(b).0$ with label 1, we also pattern match with the rules INP2A and INP2B, both resulting in 0. We are able to return the result at this stage too. Furthermore, from these two states, we can apply DBLINP, resulting in the double labels 11 and $11^{\bullet}$. Additionally from this point, we can return back to the MATCH rule. This behaviour is captured informally with the diagram below. The arrows returning to the MATCH rule denotes the returning of a transition result.

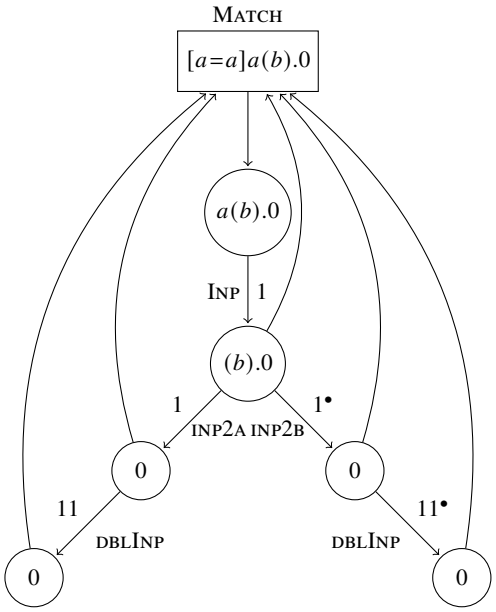


Fig. 4.1: Illustrating multiple points in the transition derivation tree where the rules can be applied.

The next transition labels that we should receive only are 11 and $11^{\bullet}$. In a practical setting, these rules become problematic and are non-ideal for implementation. Duplicate transitions may arise as a result of nested transition rule applications. This ambiguity also occurs for the rules RES, SUM, REC, and PAR. Due to this, we define a revised $\times\pi$-calculus transition relation where rules are applied only in one place of the transition derivation tree.

4.2 Revised transition relation

The source of ambiguity in the original transition relation was the decomposition of the input and output processes into intermediate stages. These intermediate stages formed final double labels with the DblInp and DblOut rules. In the revised transition relation, we completely eliminate the use of intermediate steps. Therefore, we redefine the set of labels, which consists of only the tau and double labels.

Definition 4.3. Revised labels. The set of revised labels α is defined as

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

In addition, we introduce the following definitions to help us refine the transition relation.

Definition 4.4. Label index. A register label index is defined as $\mathsf{ind}(\alpha) : \mathcal{P}(\mathbb{N})$ where

$$\begin{aligned}
\mathsf{ind}(\tau) &= \varnothing \\
\mathsf{ind}(\bar{i}j) &= \{i, j\} \\
\mathsf{ind}(\bar{i}j^{\circledast}) &= \{i, j\} \\
\mathsf{ind}(ij) &= \{i, j\} \\
\mathsf{ind}(ij^{\bullet}) &= \{i, j\}
\end{aligned}$$

$\mathsf{ind}(\alpha)$ is the set of single label indices of a label in the set of labels α. We require this due to vagueness with the notation $\alpha \neq (|\sigma|+1)$ in the Res rule. This side condition may seem clear for single labels in α. However, α contains double labels, so this notation does not clarify if it includes the first label or second label. The label index definition makes this intent more rigourous.

Definition 4.5. Label symbol. A label symbol is defined as $\mathsf{sym}(\alpha)$ where

$$\begin{aligned}
\mathsf{sym}(\tau) &= \{\tau\} \\
\mathsf{sym}(\bar{i}j) &= \{\bar{i}, j\} \\
\mathsf{sym}(\bar{i}j^{\circledast}) &= \{\bar{i}, j^{\circledast}\} \\
\mathsf{sym}(ij) &= \{i, j\} \\
\mathsf{sym}(ij^{\bullet}) &= \{i, j^{\bullet}\}
\end{aligned}$$

$\mathsf{sym}(\alpha)$ is the set of single label symbols of a label in the set of labels α which recognises the type of the label.

Finally, we define the revised $\times\pi$-calculus transition relation in table 4.2.

$$\textsc{Inp1}\quad \frac{\sigma \vdash a(b).P \xrightarrow{i} \sigma \vdash (b).P \xrightarrow{j} \sigma \vdash P\{c/b\}}{\sigma \vdash a(b).P \xrightarrow{ij} \sigma \vdash P\{c/b\}} \begin{array}{l}\sigma(i){=}a\\ \sigma(j){=}c\end{array}$$

$$\textsc{Inp2}\quad \frac{\sigma \vdash a(b).P \xrightarrow{i} \sigma \vdash (b).P \xrightarrow{j^\bullet} \sigma[j \mapsto b] \vdash P}{\sigma \vdash a(b).P \xrightarrow{ij^\bullet} \sigma[j \mapsto b] \vdash P} \begin{array}{l}\sigma(i){=}a\\ j{=}\min\{\, j \mid \sigma(j){\notin}\mathsf{fn}(P)\,\}\\ b{\notin}\mathsf{img}(\sigma)\end{array}$$

$$\textsc{Out}\quad \frac{\sigma \vdash \bar{a}b.P \xrightarrow{i} \sigma \vdash b.P \xrightarrow{j} \sigma \vdash P}{\sigma \vdash \bar{a}b.P \xrightarrow{\bar{i}j} \sigma \vdash P} \begin{array}{l}\sigma(i){=}a\\ \sigma(j){=}b\end{array}$$

$$\textsc{Res}\quad \frac{(\sigma + a) \vdash P \xrightarrow{\alpha} (\sigma' + a) \vdash P'}{\sigma \vdash \nu a.P \xrightarrow{\alpha} \sigma' \vdash \nu a.P'} \; (|\sigma|{+}1){\notin}\mathsf{ind}(\alpha)$$

$$\textsc{Open}\quad \frac{(\sigma + a) \vdash P \xrightarrow{\bar{i}j} (\sigma + a) \vdash P'}{\sigma \vdash \nu a.P \xrightarrow{\bar{i}k^{\circledast}} \sigma[k \mapsto a] \vdash P'} \begin{array}{l}i{\neq}j\\ j{=}(|\sigma|{+}1)\\ k{=}\min\{\, i \mid \sigma(i){\notin}\mathsf{fn}(P')\,\}\\ a{\notin}\mathsf{img}(\sigma)\end{array}$$

$$\textsc{Match1}\quad \frac{\sigma \vdash P \xrightarrow{\alpha} \sigma' \vdash P'}{\sigma \vdash [a = a]P \xrightarrow{\alpha} \sigma' \vdash P'} \qquad \textsc{Match2}\quad \frac{\sigma \vdash P \xrightarrow{\alpha} \sigma' \vdash P'}{\sigma \vdash [a \neq b]P \xrightarrow{\alpha} \sigma' \vdash P'}$$

$$\textsc{Rec}\quad \frac{\sigma \vdash P\{\vec{a}/\vec{b}\} \xrightarrow{\alpha} \sigma' \vdash P'}{\sigma \vdash p(\vec{a}) \xrightarrow{\alpha} \sigma' \vdash P'} \; p(\vec{b}){=}P$$

$$\textsc{Sum}\quad \frac{\sigma \vdash P \xrightarrow{\alpha} \sigma' \vdash P'}{\sigma \vdash P + Q \xrightarrow{\alpha} \sigma' \vdash P'} \qquad \textsc{Sum}_{\textsc{sym}}\quad \frac{\sigma \vdash Q \xrightarrow{\alpha} \sigma' \vdash Q'}{\sigma \vdash P + Q \xrightarrow{\alpha} \sigma' \vdash Q'}$$

$$\textsc{Par1}\quad \frac{\sigma \vdash P \xrightarrow{\alpha} \sigma \vdash P'}{\sigma \vdash P \mid Q \xrightarrow{\alpha} \sigma \vdash P' \mid Q} \; j^\bullet, j^{\circledast}{\notin}\mathsf{sym}(\alpha)$$

$$\textsc{Par1}_{\textsc{sym}}\quad \frac{\sigma \vdash Q \xrightarrow{\alpha} \sigma \vdash Q'}{\sigma \vdash P \mid Q \xrightarrow{\alpha} \sigma \vdash P \mid Q'} \; j^\bullet, j^{\circledast}{\notin}\mathsf{sym}(\alpha)$$

$$\textsc{Par2}\quad \frac{\sigma \vdash P \xrightarrow{ij^\bullet/\bar{i}j^{\circledast}} \sigma[j \mapsto b] \vdash P'}{\sigma \vdash P \mid Q \xrightarrow{ik^\bullet/\bar{i}k^{\circledast}} \sigma[k \mapsto b] \vdash P' \mid Q} \; k{=}\min\{\, j \mid \sigma(j){\notin}\mathsf{fn}(P', Q)\,\}$$

$$\textsc{Par2}_{\textsc{sym}}\quad \frac{\sigma \vdash Q \xrightarrow{ij^\bullet/\bar{i}j^{\circledast}} \sigma[j \mapsto b] \vdash Q'}{\sigma \vdash P \mid Q \xrightarrow{ik^\bullet/\bar{i}k^{\circledast}} \sigma[k \mapsto b] \vdash P \mid Q'} \; k{=}\min\{\, j \mid \sigma(j){\notin}\mathsf{fn}(P, Q')\,\}$$

$$\textsc{Comm}\quad \frac{\sigma \vdash P \xrightarrow{\bar{i}j} \sigma \vdash P' \qquad \sigma \vdash Q \xrightarrow{ij} \sigma \vdash Q'}{\sigma \vdash P \mid Q \xrightarrow{\tau} \sigma \vdash P' \mid Q'}$$

$$\textsc{Comm}_{\textsc{sym}}\quad \frac{\sigma \vdash P \xrightarrow{ij} \sigma \vdash P' \qquad \sigma \vdash Q \xrightarrow{\bar{i}j} \sigma \vdash Q'}{\sigma \vdash P \mid Q \xrightarrow{\tau} \sigma \vdash P' \mid Q'}$$

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

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

Tab. 4.2: The revised transition relation for the $\times\pi$-calculus.

4.3 Transition relation structure

4.3.1 Data types

We define some structs to represent the configurations, registers, and labels.

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

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

type Label struct {
    Symbol  Symbol
    Symbol2 Symbol
}

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

Listing 4.1: Data types to represent configurations, registers, and labels.

A configuration consists of a process, a register assignment, and a label taken to reach this configuration. The register assignment holds a map which maps a label integer to a name string. The label, representing an element in set α, is only a single type, which is not polymorphised for practical convenience. The symbol struct holds the label symbol type and label value. The symbol type represents the element types in $\mathsf{sym}(\alpha)$.

4.3.2 Register initialisation

The register assignment of the root configuration is initialised by populating the registers with the free names of the undeclared and declared processes. n number of free names are sorted lexicographically and are assigned a label from 1 to n. The undeclared process, already alpha-converted, is passed through a function which traverses the AST and finds the free names. For declared processes, we alpha-convert the tree and find the free names. For recursive process calls, a set of visited processes prevents infinite cycles.

The `Size` attribute of the register assignment is initialised by default to be virtually unlimited. In practice, this is 2^{30} (half of the positives in int32). The reason for the finiteness of the registers is due to Tzevelekos' definition of FRA's. There is also a practical purpose to having registers of limited size, which we will see in the transition rules.

4.3.3 Top-level transition function

`trans()` is a top-level function which performs all the transition rules by pattern matching the first element in the process. Its function signature is

```
func trans(conf Configuration) []Configuration
```

which takes in a configuration and returns a list of all possible next configurations.

4.4 Transition rules

We shall now examine the rules thoroughly and how they differ from the original transition rules. Transition rules are described by example for better understanding. Implementation details can be extrapolated because the examples themselves describe well the algorithm. Assume that all expressions given have undergone alpha-conversion, and so the bound names correspond uniquely to their respective binders.

4.4.1 Input

$$\textsc{Inp1}\quad \frac{\sigma \vdash a(b).P \xrightarrow{i} \sigma \vdash (b).P \xrightarrow{j} \sigma \vdash P\{c/b\}}{\sigma \vdash a(b).P \xrightarrow{ij} \sigma \vdash P\{c/b\}} \begin{array}{l}\sigma(i)=a\\ \sigma(j)=c\end{array}$$

$$\textsc{Inp2}\quad \frac{\sigma \vdash a(b).P \xrightarrow{i} \sigma \vdash (b).P \xrightarrow{j^\bullet} \sigma[j \mapsto b] \vdash P}{\sigma \vdash a(b).P \xrightarrow{ij^\bullet} \sigma[j \mapsto b] \vdash P} \begin{array}{l}\sigma(i)=a\\ j=\min\{\, j \mid \sigma(j) \notin \mathsf{fn}(P) \,\}\\ b \notin \mathsf{img}(\sigma)\end{array}$$

Tab. 4.3: The revised $\times\pi$-calculus transition relation for input.

$$\textsc{Inp1}\quad \frac{}{\sigma \vdash a(b).P \xrightarrow{i} \sigma \vdash (b).P} \; \sigma(i)=a$$

$$\textsc{Inp2a}\quad \frac{}{\sigma \vdash (b).P \xrightarrow{i} \sigma \vdash P\{a/b\}} \; \sigma(i)=a$$

$$\textsc{Inp2b}\quad \frac{}{\sigma \vdash (b).P \xrightarrow{i^\bullet} \sigma[i \mapsto b] \vdash P} \begin{array}{l} i=\min\{\, i \mid \sigma(i) \notin \mathsf{fn}(P) \,\}\\ b \notin \mathsf{img}(\sigma)\end{array}$$

$$\textsc{DblInp}\quad \frac{\sigma \vdash P \xrightarrow{i} \sigma \vdash P_{\mathsf{inp}} \xrightarrow{j/j^\bullet} \sigma' \vdash P'}{\sigma \vdash P \xrightarrow{ij/ij^\bullet} \sigma' \vdash P'}$$

Tab. 4.4: The original $\times\pi$-calculus transition relation for input.

The two revised Inp rules replace the four original Inp and DblInp rules.

Example 4.1. Input transition rule. Consider the configuration

$$\{(1, a), (2, b)\} \vdash a(x).\bar{b}\langle x\rangle.0$$

From the original Inp1, this rule has no premise. As a result, we can directly take the transition to $(x).\bar{b}\langle x\rangle.0$ with label i. From the side condition, label i is defined as $\sigma(i) = a$, meaning that i is a label corresponding to the channel name a in the registers σ. In this case, $\sigma(1) = a$, so $i = 1$.

From $(x).\bar{b}\langle x\rangle.0$, we can take both Inp2a and Inp2b. In Inp2a, the input intermediate stage takes label i to $\bar{b}\langle x\rangle.0$. From the side condition, i corresponds to every name in the register assignment, so we get two labels 1 and 2 from this rule, corresponding to a and b, respectively. The conclusion also states that we must substitute the x's in $\bar{b}\langle x\rangle.0$ with the label name. So the result becomes $\xrightarrow{1} \sigma \vdash \bar{b}\langle a\rangle.0$ and $\xrightarrow{2} \sigma \vdash \bar{b}\langle b\rangle.0$, with σ remaining the same. Inp2a represents the resultant configurations when the name received on the channel is a name in the registers.

In Inp2b, from $(x).\bar{b}\langle x\rangle.0$, we take the transition $i^\bullet$ to $\bar{b}\langle x\rangle.0$. Notice the update on the registers $\sigma[i \mapsto b]$. i is the minimum label of the register assignment where the name corresponding to that label does not occur as a free name in $\bar{b}\langle x\rangle.0$. We see that a does not appear in P, so the result becomes $\xrightarrow{1^\bullet} \{(1, x), (2, b)\} \vdash \bar{b}\langle x\rangle.0$. If a and b did appear in P, then the result becomes $\xrightarrow{3^\bullet} \{(1, a), (2, b), (3, x)\} \vdash P$, where we use the next available label in σ. $b \notin \mathsf{img}(\sigma)$ means to use a name which does not exist in the register assignment. Since we alpha-converted the bound names to a generated name to their binder, this cannot occur, and b remains always unique.

From the above Inp rules, we get $P \xrightarrow{1} P' \xrightarrow{1} Q$, $P \xrightarrow{1} P' \xrightarrow{2} Q'$, and $P \xrightarrow{1} P' \xrightarrow{1^\bullet} Q''$, where $P, P', Q, Q', Q'' \in O(\hat{K})$. These transitions match with the premise of DblInp, and as a result form $P \xrightarrow{11} Q$, $P \xrightarrow{12} Q'$, and $P \xrightarrow{11^\bullet} Q''$.

The revised transition relation performs the same steps but combines the original rules Inp1, Inp2a, and DblInp to form Inp1 and Inp1, Inp2b, and DblInp to form Inp2.

4.4.2 Output

$$\textsc{Out}\quad \frac{\sigma \vdash \bar{a}b.P \xrightarrow{i} \sigma \vdash b.P \xrightarrow{j} \sigma \vdash P}{\sigma \vdash \bar{a}b.P \xrightarrow{\bar{i}j} \sigma \vdash P}\ \begin{matrix}\sigma(i)=a\\ \sigma(j)=b\end{matrix}$$

Tab. 4.5: The revised $\times\pi$-calculus transition relation for output.

$$\textsc{Out1}\quad \frac{}{\sigma \vdash \bar{a}b.P \xrightarrow{i} \sigma \vdash b.P}\ \sigma(i)=a \qquad \textsc{Out2}\quad \frac{}{\sigma \vdash b.P \xrightarrow{i} \sigma \vdash P}\ \sigma(i)=b$$

$$\textsc{DblOut}\quad \frac{\sigma \vdash P \xrightarrow{i} \sigma \vdash P_{\mathsf{out}} \xrightarrow{j/j^{\circledast}} \sigma' \vdash P'}{\sigma \vdash P \xrightarrow{\bar{i}j/\bar{i}j^{\circledast}} \sigma' \vdash P'}$$

Tab. 4.6: The original $\times\pi$-calculus transition relation for output.

Example 4.2. Output transition rule. Consider the configuration $\sigma \vdash P$

$$\{(1,a),(2,b)\} \vdash \bar{a}\langle b\rangle.0$$

From the original Out1, we take the transition $\xrightarrow{1} \sigma \vdash \langle b\rangle.0$, with 1 corresponding to the channel name a. Consequently, we apply the rule Out2, where we get $\sigma \vdash \langle b\rangle.0 \xrightarrow{2} \sigma \vdash 0$. The label 2 corresponds to the output name b. With the transition $\sigma \vdash \bar{a}\langle b\rangle.0 \xrightarrow{1} \sigma \vdash \langle b\rangle.0 \xrightarrow{2} \sigma \vdash 0$, we apply DblOut to get $\sigma \vdash \bar{a}\langle b\rangle.0 \xrightarrow{\bar{1}2} \sigma \vdash 0$. This can be read as on the channel name with label 1 (corresponding to the name a), output the name with label 2 (corresponding to the name b).

In the revised Out rule, the actions described above are combined into one rule.

4.4.3 Restriction

$$\textsc{Res}\quad \frac{(\sigma + a) \vdash P \xrightarrow{\alpha} (\sigma' + a) \vdash P'}{\sigma \vdash \nu a.P \xrightarrow{\alpha} \sigma' \vdash \nu a.P'}\ (|\sigma|+1) \notin \mathsf{ind}(\alpha)$$

Tab. 4.7: The revised $\times\pi$-calculus transition rule for restriction.

$$\textsc{Res}\quad \frac{(\sigma + a) \vdash \hat{P} \xrightarrow{\alpha} (\sigma' + a) \vdash \hat{P}'}{\sigma \vdash \nu a.\hat{P} \xrightarrow{\alpha} \sigma' \vdash \nu a.\hat{P}'}\ \alpha \neq (|\sigma|)+1)$$

Tab. 4.8: The original $\times\pi$-calculus transition rule for restriction.

In the revised Res rule, there are some small differences compared with the original rule. Note the replacement $\hat{P}$ with P, which now excludes intermediate stages. α is redefined to eliminate single label transitions. The side condition provides a more rigorous definition than the original.

Note the syntax $(\sigma + a)$. This is defined as $\sigma + v = \sigma \cup \{(|\sigma|+1, v)\}$. This means placing a name at the register assignment size+1'th label of the registers. Register assignment size refers to the maximum size of the registers, not the number of existing label-name pairs in the register assignment. For implementing this operation, by default we previously specified the size of the registers to be extremely large (2^{30}). This specifies a label index to place a name and that populating all the register slots up to this point becomes unlikely.

Example 4.3. Restriction transition rule. Consider the configuration

$$\{(1, a)\} \vdash \nu b.a(x).0$$

and suppose the size of the registers is 100. The restriction element pattern matches with the revised Res rule. We perform the premise and apply the $(\sigma + a)$ operation. The bound name b is placed at the label 101, resulting in $\{(1, a), (101, b)\}$. With this register assignment, we gather the next transitions $\xrightarrow{11} \{(1, a), (101, b)\} \vdash 0$ and $\xrightarrow{11^\bullet} \{(1, x), (101, b)\} \vdash 0$. In the premise, note that the registers σ may change, hence σ'. This occurs for the second result. In the conclusion, we must remove the name added at the label register assignment size+1, but retain the altered labels of σ'. We do this and also decrement the register assignment size by one, which allows for keeping track of nested restrictions. As a result, the conclusions are $\xrightarrow{11} \{(1, a)\} \vdash \nu b.0$ and $\xrightarrow{11^\bullet} \{(1, x)\} \vdash \nu b.0$.

Consider also the configuration

$$\{(1, a)\} \vdash \nu b.\bar{b}\langle a\rangle.0$$

From inspection, we see that this expression is 'stuck'. This is because the name b is bound and we attempt to send a name on this bound channel. As a result, we expect no next transitions. The side condition of the Res rule captures this behaviour and prevents these potential transitions. We attempt the premise of the rule and find the transition $\xrightarrow{\overline{101}\,1} \{(1, x), (101, b)\} \vdash 0$. However, the side condition states that neither the first nor second label should be the bound name label 101. Therefore, we discard our only transition, which results in no conclusions. When the second label is the bound name, the Open rule takes effect.

4.4.4 Open

$$\textsc{Open}\quad \frac{(\sigma + a) \vdash P \xrightarrow{\bar{i}j} (\sigma + a) \vdash P'}{\sigma \vdash \nu a.P \xrightarrow{\bar{i}k^{\circledast}} \sigma[k \mapsto a] \vdash P'} \begin{array}{l} i \neq j \\ j = (|\sigma| + 1) \\ k = \min\{\, i \mid \sigma(i) \notin \mathsf{fn}(P') \,\} \\ a \notin \mathsf{img}(\sigma) \end{array}$$

Tab. 4.9: The revised $\times\pi$-calculus transition rule for open.

$$\textsc{Open}\quad \frac{\sigma[i \mapsto a] \vdash P_{\mathsf{out}} \xrightarrow{i} \sigma[i \mapsto a] \vdash P}{\sigma \vdash \nu a.P_{\mathsf{out}} \xrightarrow{i^{\circledast}} \sigma[i \mapsto a] \vdash P} \begin{array}{l} i = \min\{\, i \mid \sigma(i) \notin \mathsf{fn}(P) \,\} \\ a \notin \mathsf{img}(\sigma) \end{array}$$

Tab. 4.10: The original $\times\pi$-calculus transition rule for open.

The Open rule captures the transition of fresh outputs where a bound name of a restriction is sent as an output name. In the revised rule, we make a number of changes. The intermediate stage of P_{out} cannot occur anymore and the intended transition is converted to a double label. Instead of placing the bound name at the minimum available label $\sigma[i \mapsto a]$ at the premise, we place it at the register assignment size+1 label $(\sigma + a)$, like with Res. Also, instead of representing the bound name transition implicitly in the rule, we introduce the side condition such that the second label $j = (|\sigma| + 1)$. In addition, $i \neq j$, so that the channel name cannot be the bound name. Finally, if we do fulfil the premise, we place this bound name at the minimum available label $\sigma[k \mapsto a]$. Because of alpha-conversion, we do not need to take any actions for $a \notin \mathsf{img}(\sigma)$.

Example 4.4. Open transition rule. Consider the configuration $\sigma \vdash \nu a.P$

$$\{(1, a)\} \vdash \nu b.\bar{a}\langle b \rangle.0$$

and suppose the size of the registers is 100. We perform the operation $(\sigma + a)$ and gather the transition for P, resulting in $\xrightarrow{\bar{1}\ 101} \{(1, a), (101, b)\} \vdash 0$. We see that the this is an output with the second label corresponding to the bound name b, i.e., it sends the bound name b on channel a. The premise is fulfilled and we form the conclusion. We take the starting σ and place the bound name at the minimum label where that label's name is not used in P'. This label also replaces the second label of the conclusion and is converted to a fresh output label symbol. As a result, we form the conclusion $\xrightarrow{11^{\circledast}} \{(1, b)\} \vdash 0$. The restriction disappears as it is now a free name.

4.4.5 Match

$$\textsc{Match1}\quad \frac{\sigma \vdash P \xrightarrow{\alpha} \sigma' \vdash P'}{\sigma \vdash [a = a]P \xrightarrow{\alpha} \sigma' \vdash P'} \qquad \textsc{Match2}\quad \frac{\sigma \vdash P \xrightarrow{\alpha} \sigma' \vdash P'}{\sigma \vdash [a \neq b]P \xrightarrow{\alpha} \sigma' \vdash P'}$$

Tab. 4.11: The revised $\times\pi$-calculus transition rule for match.

$$\textsc{Match}\quad \frac{\sigma \vdash P \xrightarrow{\alpha} \sigma \vdash \hat{P}'}{\sigma \vdash [a = a]P \xrightarrow{\alpha} \sigma \vdash \hat{P}'}$$

Tab. 4.12: The original $\times\pi$-calculus transition relation for match.

The revised Match1 rule differs subtly from the original. $\hat{P}$ is replaced with P, which excludes the intermediate input/outputs. σ becomes σ', meaning that in the revised rule, the registers may be subject to change. Also note that α is redefined, which includes only double transitions and tau steps only.

The Match2 rule is introduced to accommodate for the inequality construct $[a{\neq}b]P$, which was not originally defined in the $\times\pi$-calculus language by Tzevelekos. Its rule is identical to Match1, except that the condition is inequality rather than equality.

Example 4.5. Match transition rule. Consider the configuration

$$\{(1, a)\} \vdash a(x).[a{=}x]\nu b.\bar{a}\langle b\rangle.0$$

We take $\xrightarrow{11} \{(1, a)\} \vdash [a{=}a]\nu b.\bar{a}\langle b\rangle.0$ from Inp1 and $\xrightarrow{12^{\bullet}} \{(1, a), (1, x)\} \vdash [a{=}x]\nu b.\bar{a}\langle b\rangle.0$ from Inp2. For the first result where the bound name x is substituted with a, the Match1 rule to can proceed. In contrast, for the second result, the name x becomes a fresh input, thus the equality $a{=}x$ does not hold, and so the expression cannot be reduced further.

From the first result, we can therefore apply the Match1 rule, resulting in $\xrightarrow{\bar{1}1^{\circledast}} \{(1, b)\} \vdash 0$ from Open. This result shows the rationale behind the σ' in the revised rule, as σ may change from the starting registers, as in this case. For the second result, if the construct is $[a{\neq}x]P$, then we would be able to continue by the Match2 rule.

4.4.6 Recursion

$$\textsc{Rec} \quad \frac{\sigma \vdash P\{\vec{a}/\vec{b}\} \xrightarrow{\alpha} \sigma' \vdash P'}{\sigma \vdash p(\vec{a}) \xrightarrow{\alpha} \sigma' \vdash P'} \; p(\vec{b})=P$$

Tab. 4.13: The revised $\times\pi$-calculus transition rule for recursion.

$$\textsc{Rec} \quad \frac{\sigma \vdash P\{\vec{a}/\vec{b}\} \xrightarrow{\alpha} \sigma \vdash \hat{P}'}{\sigma \vdash p(\vec{a}) \xrightarrow{\alpha} \sigma \vdash \hat{P}'} \; p(\vec{b})=P$$

Tab. 4.14: The original $\times\pi$-calculus transition rule for recursion.

Note the changes in P', σ', and the redefined α.

Example 4.6. Recursion transition rule. Consider the definition and configuration

$$p(x, y) = \bar{x}\langle y\rangle.0$$
$$\{(1, a)\} \vdash a(b).p(a, b)$$

We take $\xrightarrow{11} \{(1, a)\} \vdash p(a, a)$ from Inp1 and $\xrightarrow{12^\bullet} \{(1, a), (2, b)\} \vdash p(a, b)$ from Inp2. When we encounter a process call, we replace the names $\vec{b}$ of the process definition with the names $\vec{a}$ of the process call, similar to function parameters and arguments in programming languages. Unfolding the definition, the first result's process resolves to $\bar{a}\langle a\rangle.0$ and the second to $\bar{a}\langle b\rangle.0$. The next transition from the first result gathers $\xrightarrow{\bar{1}1} \{(1, a)\} \vdash 0$ and from the second $\xrightarrow{\bar{1}2} \{(1, a), (2, b)\} \vdash 0$.

4.4.7 Sum

$$\textsc{Sum} \quad \frac{\sigma \vdash P \xrightarrow{\alpha} \sigma' \vdash P'}{\sigma \vdash P + Q \xrightarrow{\alpha} \sigma' \vdash P'} \qquad \textsc{Sum}_{\textsc{sym}} \quad \frac{\sigma \vdash Q \xrightarrow{\alpha} \sigma' \vdash Q'}{\sigma \vdash P + Q \xrightarrow{\alpha} \sigma' \vdash Q'}$$

Tab. 4.15: The revised $\times\pi$-calculus transition relation for summation.

$$\textsc{Sum} \quad \frac{\sigma \vdash P \xrightarrow{\alpha} \sigma \vdash \hat{P}'}{\sigma \vdash P + Q \xrightarrow{\alpha} \sigma \vdash \hat{P}'} \qquad \textsc{Sum}_{\textsc{sym}} \quad \frac{\sigma \vdash Q \xrightarrow{\alpha} \sigma \vdash \hat{Q}'}{\sigma \vdash P + Q \xrightarrow{\alpha} \sigma \vdash \hat{Q}'}$$

Tab. 4.16: The original $\times\pi$-calculus transition relation for summation.

The summation transition relation comprises of both the SUM rule and its symmetric counterpart SUM_{SYM}, representing the selection of the left-hand side process and right-hand side process, respectively.

Example 4.7. Sum transition rule. Consider the configuration with process $P + Q$,

$$\{(1, a), (2, b)\} \vdash \bar{a}\langle b\rangle.0 + a(x).0$$

The SUM rule attempts the premise, which gathers the transition $\xrightarrow{\bar{1}2} \{(1, a), (2, b)\} \vdash 0$. This transition is also the conclusion which discards the possibility of taking the process Q. In addition, the SUM_{SYM} rule also applies, which performs the same rule for the process Q. We get $\xrightarrow{11} \{(1, a), (2, b)\} \vdash 0$, $\xrightarrow{12} \{(1, a), (2, b)\} \vdash 0$, and $\xrightarrow{11^\bullet} \{(1, x), (1, b)\} \vdash 0$. Note that the registers change to σ' for the last result, which is present only in the revised transition rule.

4.4.8 Parallel

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

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

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

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

Tab. 4.17: The revised $\times\pi$-calculus transition relation for composition.

$$\text{PAR1} \quad \frac{\sigma \vdash P \xrightarrow{\alpha} \sigma \vdash \hat{P}'}{\sigma \vdash P \mid Q \xrightarrow{\alpha} \sigma \vdash \hat{P}' \mid Q}$$

$$\text{PAR1}_{\text{SYM}} \quad \frac{\sigma \vdash Q \xrightarrow{\alpha} \sigma \vdash \hat{Q}'}{\sigma \vdash P \mid Q \xrightarrow{\alpha} \sigma \vdash P \mid \hat{Q}'}$$

$$\text{PAR2} \quad \frac{\sigma \vdash \hat{P} \xrightarrow{i^\bullet/i^\circledast} \sigma[i \mapsto b] \vdash P'}{\sigma \vdash \hat{P} \mid Q \xrightarrow{j^\bullet/j^\circledast} \sigma[j \mapsto b] \vdash P' \mid Q} \; j = \min\{\, j \mid \sigma(j) \notin \mathsf{fn}(P', Q) \,\}$$

$$\text{PAR2}_{\text{SYM}} \quad \frac{\sigma \vdash \hat{Q} \xrightarrow{i^\bullet/i^\circledast} \sigma[i \mapsto b] \vdash Q'}{\sigma \vdash P \mid \hat{Q} \xrightarrow{j^\bullet/j^\circledast} \sigma[j \mapsto b] \vdash P \mid Q'} \; j = \min\{\, j \mid \sigma(j) \notin \mathsf{fn}(P, Q') \,\}$$

Tab. 4.18: The original $\times\pi$-calculus transition relation for composition.

Two rules Par1 and Par2, and their symmetric counterparts, are required to deal with the fresh input/output cases. In the revised rule, we make it explicit in Par1 that the rule only deals with non-fresh input/outputs. In contrast, in the original Par1, this non-fresh input/output rule is implicitly captured by the unchanged σ, because only fresh input/outputs can change the registers. So, for practicality and clarity, the $j^\bullet, j^\circledast \notin \mathsf{sym}(\alpha)$ side condition was added to the revised Par1.

The Par2 rule was revised to take into account only double fresh input/outputs, since single labels are eliminated. The differences are slight. In the revised rule, we check the second label to see if it's a fresh input/output, and if so, then move its label to the minimum label as per the side condition, which is performed also in the original rule.

Example 4.8. Parallel transition rule. Consider the configuration with process $P \mid Q$,

$$\{(1, a), (2, b)\} \vdash \bar{a}\langle b\rangle.0 \mid b(x).0$$

The starting configuration of the premise of the revised Par1 and Par2 do not differ. The premise's result of the two rules are disjoint. As a result, we can gather the results and see which Par rule they match with. Taking the premise transitions of P, the result is $\xrightarrow{\bar{1}2} \{(1, a), (2, b)\} \vdash 0$. We see that this is not a fresh input/output, so we can take the conclusion of Par1 using this transition. As a result, we form a parallel with Q and get $\xrightarrow{\bar{1}2} \{(1, a), (2, b)\} \vdash 0 \mid b(x).0$.

Performing the premise of the symmetric rules Par1_{SYM} and Par2_{SYM} on the other process Q, we get $\xrightarrow{21} \{(1, a), (2, b)\} \vdash 0$, $\xrightarrow{22} \{(1, a), (2, b)\} \vdash 0$, and $\xrightarrow{23^\bullet} \{(1, a), (2, b), (3, x)\} \vdash 0$. The first and second result matches with Par1_{SYM}, resulting in $\xrightarrow{21} \{(1, a), (2, b)\} \vdash \bar{a}\langle b\rangle.0 \mid 0$ and $\xrightarrow{22} \{(1, a), (2, b)\} \vdash \bar{a}\langle b\rangle.0 \mid 0$. However, for the third result, we can see that the second label contains a fresh input, thus it matches with the Par2_{SYM} rule. From the resultant configuration of the premise, the registers become $\sigma[j \mapsto b]$. In his case, $j = 3$, which maps to the name x. The conclusion maps this name to a label k, as specified by the side condition, which states that k is the minimum label where the name at that label does not appear as a free name in P or Q'. This means that we use the minimum unused label. However, in this case, we see that the names a and b appear in P, so the minimum label is the next available label, which is 3. So the conclusion is $\xrightarrow{23^\bullet} \{(1, a), (2, b), (3, x)\} \vdash \bar{a}\langle b\rangle.0 \mid 0$. If a did not appear in P or Q', we would get the label $21^\bullet$ and register assignment $\{(1, x), (2, b)\}$, where the name a in label 1 is overwritten.

4.4.9 Communication

$$\textsc{Comm} \quad \frac{\sigma \vdash P \xrightarrow{\bar{i}j} \sigma \vdash P' \qquad \sigma \vdash Q \xrightarrow{ij} \sigma \vdash Q'}{\sigma \vdash P \mid Q \xrightarrow{\tau} \sigma \vdash P' \mid Q'}$$

$$\textsc{Comm}_{\textsc{sym}} \quad \frac{\sigma \vdash P \xrightarrow{ij} \sigma \vdash P' \qquad \sigma \vdash Q \xrightarrow{\bar{i}j} \sigma \vdash Q'}{\sigma \vdash P \mid Q \xrightarrow{\tau} \sigma \vdash P' \mid Q'}$$

Tab. 4.19: The $\times\pi$-calculus transition relation (both original and revised) for communication.

The revised communication rule is identical to the original rule due to the exclusive use of double transitions and taus. The communication rule captures the sending and receiving of a variable along channels of the same name in parallel processes. Notice that there are two premises for Comm, one each for the left and right-hand side processes.

Example 4.9. Communication transition rule. Consider the configuration with process $P \mid Q$,

$$\{(1, a), (2, b)\} \vdash \bar{a}\langle b\rangle.0 \mid a(x).0$$

By taking the transitions of P, we get $\xrightarrow{\bar{1}2} \{(1, a), (2, b)\} \vdash 0$. The transitions of Q are $\xrightarrow{11} \{(1, a), (2, b)\} \vdash 0$, $\xrightarrow{12} \{(1, a), (2, b)\} \vdash 0$, and $\xrightarrow{11^\bullet} \{(1, x), (2, b)\} \vdash 0$. The Comm rule states that the double output labels of the left-hand side P must match with the double input labels of the right-hand side Q, i.e., $\bar{i} = i \wedge j = j$. In this case, we see the paired labels $\bar{1}2$ and 12. As a result, we can take the tau step conclusion, resulting in $\xrightarrow{\tau} \{(1, a), (2, b)\} \vdash 0 \mid 0$. The $\textsc{Comm}_{\textsc{sym}}$ rule performs the equivalent steps, except the input and output sides are swapped.

4.4.10 Close

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

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

Tab. 4.20: The $\times\pi$-calculus transition relation (both original and revised) for close.

The close rule is no different for the revised transition relation. The rule captures scope extension of a restriction in one process to include two processes.

Note the syntax $(\sharp + \sigma)$. This is defined as $v + \sigma = \{(1, v)\} \cup \{(i{+}1, v') \mid (i, v') \in \sigma\}$. This means incrementing all the labels by one in the registers while retaining a mapping to their names. Then, we place the special name $\sharp$ at label 1.

Example 4.10. Close transition rule. Consider the configuration with process $P \mid Q$,

$$\{(1, a)\} \vdash \nu b.\bar{a}\langle b\rangle.0 \mid a(c).0$$

We perform the operation $(\sharp + \sigma)$ on the registers and find the transitions of P. The result is $\xrightarrow{\bar{2}1^{\circledast}} \{(1, b), (2, a)\} \vdash 0$ from Open. Note the position of $(2, a)$ because we shifted the label-names right by one. We fulfil the premise's transition criteria of $\bar{i}1^{\circledast}$. Conversely, we perform the $(\sharp + \sigma)$ operation and find the transitions of Q. We gather $\xrightarrow{21} \{(1, \sharp), (2, a)\} \vdash 0$ and $\xrightarrow{22} \{(1, \sharp), (2, a)\} \vdash 0$ from Inp1, and $\xrightarrow{21^{\bullet}} \{(1, c), (2, a)\} \vdash 0$ from Inp2. Because of the $i1^{\bullet}$ condition for Q, we can disregard the first two results.

This leaves us with the matching premises $\xrightarrow{\bar{2}1^{\circledast}} \{(1, b), (2, a)\} \vdash 0$ and $\xrightarrow{21^{\bullet}} \{(1, c), (2, a)\} \vdash 0$. Note that that the second result does not match exactly with the premise due to a different name c at label 1 in the register assignment. The rule states that the both the resultant registers should be $(b + \sigma)$. However, it is expected that the names are different because it is impossible to have a fresh output and fresh input to be of the same name due to alpha-conversion. To amend this, we substitute the name c with b, so Q' becomes $Q'\{b/c\}$. This is allowed because any fresh name can be chosen for fresh input. In the Close rule, this subtle step is implicit. As a result from these two premises, we can form the conclusion $\xrightarrow{\tau} \{(1, b), (2, a)\} \vdash \nu b.(0 \mid 0)$.

Congruence 5

After applying the transition rules, we find that there are seemingly different states, or configurations, which can be considered equivalent.

5.1 Structural congruence

Definition 5.1. Structural congruence. [1, p. 31, definition 4.7]. Structural congruence, written $\equiv$, is the process congruence over $\mathcal{P}$ determined by the following equations:

(1) Change of bound names (alpha-conversion)
(2) Reordering of terms in a summation
(3) $P|0 \equiv P$, $P|Q \equiv Q|P$, $P|(Q|R) \equiv (P|Q)|R$
(4) $\nu a.(P|Q) \equiv P|\nu a.Q$ if $a \notin \mathsf{fn}(P)$
$\nu a.0 \equiv 0$, $\nu a.\nu b.P \equiv \nu b.\nu a.P$
(5) $A\langle\vec{b}\rangle \equiv \{\vec{b}/\vec{a}\}P_A$ if $A(\vec{a}) \stackrel{\text{def}}{=} P_A$

5.2 Configuration congruence

Structural congruence only deals with processes. However, in our case, each state comprises of both a process and a register assignment. Therefore, we extend this notion of congruence in order to accommodate the definition of configurations in the $\times\pi$-calculus. A number of definitions are introduced to build up the definition of *configuration congruence*.

Definition 5.2. Well-formed configuration. A configuration $\sigma \vdash P \in O(K)$ is well-formed when

(1) σ is a register assignment, and
(2) $\forall a \in \mathsf{fn}(P),\ \exists i : \sigma(i) = a$

σ is the register assignment from the set of register assignments Reg_n as defined by FRA's, with register size of size n. There must not be any duplicate names in the registers. For all the free names in the process P, there exists a label in the register assignment corresponding to that name. Fulfilling these criteria constitutes to a valid or well-formed configuration $\sigma \vdash P$.

Definition 5.3. Permutation. A permutation is a function $\pi : \mathcal{N} \rightleftarrows \mathcal{N}$. π is a total bijective mapping between names, where $\mathcal{N}$ is the set of free names.

The notion of a permutation is formalised by De Vries and Koutavas [25, p. 6] in the paper *Locally Nameless Permutation Types*. In this case, we use a permutation to bijectively map free names. For example, the configurations $\{(1, a)\} \vdash \bar{a}\langle a\rangle.0$ and $\{(1, b)\} \vdash \bar{b}\langle b\rangle.0$ are equivalent in all but the free name. However, for the second configuration, if we have the permutation $\pi(b) = a$, then the process becomes $\{(1, a)\} \vdash \bar{a}\langle a\rangle.0$, which matches with the first configuration. We formally define permutations for registers and processes.

Definition 5.4. Permuted registers. A permutation π on register assignment σ is defined as

$$\pi\sigma = \{(i, \pi(a)) \mid \sigma(i) = a\}$$

This applies the permutation π on all names in the register assignment σ.

Definition 5.5. Permuted process. A permutation π on process P is defined as πP, where

$$\begin{aligned}
&\pi a(b).P = \pi(a)(b).\pi P \\
&\pi\bar{a}\langle b\rangle.P = \overline{\pi(a)}\langle\pi(b)\rangle.\pi P \\
&\pi[a{=}b]P = [\pi(a){=}\pi(b)]\pi P \\
&\pi[a{\neq}b]P = [\pi(a){\neq}\pi(b)]\pi P \\
&\pi\nu a.P = \nu a.\pi P \\
&\pi(P + Q) = \pi P + \pi Q \\
&\pi(P \mid Q) = \pi P \mid \pi Q \\
&\pi p(\vec{a}) = p(\{\pi(a) \mid a \in \vec{a}\}) \\
&\pi 0 = 0
\end{aligned}$$

This applies the permutation π on all $\times\pi$-calculus constructs in the process P. Note that the bound names in input and restriction are left untouched. Since bound names are placeholders, these can be represented by any name.

Definition 5.6. Permuted configuration. A permutation π on configuration $\sigma \vdash P$ is defined as $\pi(\sigma \vdash P) = \pi\sigma \vdash \pi P$.

This applies the permutation π on the configuration $\sigma \vdash P$. Finally, from this definition, we introduce configuration congruence.

Definition 5.7. Configuration congruence. Two configurations $\sigma \vdash P$ and $\sigma' \vdash P'$ are configuration congruent, written $\sigma \vdash P \equiv \sigma' \vdash P'$, if $\exists \pi$ such that

(1) $\sigma = \pi\sigma'$, and

(2) $P \equiv \pi P'$

This states that if there is a permutation π which matches the names of $\sigma' \vdash P'$ to the names of $\sigma \vdash P$, then we can conclude that the two configurations are equivalent. From this definition, we can derive some interesting properties.

Lemma 5.1. $\sigma \vdash P \equiv \sigma' \vdash P'$ implies $\sigma \vdash P \overset{\times\pi}{\sim} \sigma' \vdash P'$, where $\overset{\times\pi}{\sim}$ is an $\times\pi$-bisimulation as defined by Tzevelekos [5, p. 303, definition 33].

If two configurations are congruent, then it is no surprise that they are bisimilar.

Lemma 5.2. $\sigma \vdash P \equiv \sigma' \vdash P'$ implies $\pi \cdot \pi \cdot P = P$.

Applying the permutation π on P results in πP which matches with the names of P'. Applying the same permutation π on πP results in $\pi\pi P$ which is the same as $\pi P'$, or P.

Corollary 5.1. $\sigma \vdash P \equiv \sigma' \vdash P'$ implies $P = \pi P' \Rightarrow P' = \pi P$.

If a permutation π exists for P' matching P, then there also exists a permutation for P matching P'.

Lemma 5.2 and corollary 5.1 can also be extended to include exclusively the registers σ and also the entire configuration $\sigma \vdash P$.

5.3 Normalisation

From these definitions, the goal is to convert a configuration $\sigma \vdash P$ into a normal form, so that equivalent configurations can be detected.

5.3.1 Configuration key

The practical approach to assessing for configuration congruence is to stringify, or pretty-print, the registers and process and concatenate the two strings. Obtaining the key of a configuration $\sigma \vdash P$ is

$$\mathsf{key}(\sigma \vdash P) = \mathsf{str}(\sigma) + \mathsf{str}(P)$$

where $\mathsf{str}()$ stringifies the registers or process, $+$ concatenates two strings, and $\mathsf{key}()$ is a function which gets the configuration key. The key is the unique identifier of the configuration and can be thought of as a fingerprint or hash.

5.3.2 Free name normalisation

The root configuration $\sigma \vdash P$ is initialised to a normalised register assignment and process. Normalising the names in the registers σ consist of applying a permutation π to the names. This is achieved by the following definition.

Definition 5.8. Normalised registers. A register assignment σ of size n is normalised when a function π is applied on σ, which is defined as

$$\pi\sigma = \overrightarrow{(i, a_i)} = \overrightarrow{(j, a_j)} \cup \{(i, \#i) \mid i \leq n, i \notin \{\vec{j}\}, \#i \notin \{\vec{a_j}\}\}$$

A normalised register assignment is one where all the names in the registers are renamed to a generated free name of the form $\#1, \#2, ..., \#n$, $n \in \mathbb{N}$. For example, the normalised registers of $\{(1, a), (2, b), (3, x)\}$ is $\{(1, \#1), (2, \#2), (3, \#3)\}$. The initialisation of a normalised register assignment becomes useful for detecting equivalent configurations in fresh input/outputs and standardises the free names.

Following from the definition of a well-formed configuration, applying the above procedure also implies that the names in process P are renamed to the normalised name.

5.3.3 Bound and fresh name normalisation

The need to normalise bound and fresh names is best explained through an example.

Example 5.1. Bound and fresh name normalisation. Consider the configuration

$$\{(1, a)\} \vdash a(x).0 + a(y).0$$

After alpha-conversion, the process becomes $a(x\_0).0 + a(y\_1).0$. After the SUM rule is applied, the result for the left-hand side process is $\xrightarrow{11} \{(1, a)\} \vdash 0$ and $\xrightarrow{11^\bullet} \{(1, x\_0)\} \vdash 0$. The result for the right-hand side process is $\xrightarrow{11} \{(1, a)\} \vdash 0$ and $\xrightarrow{11^\bullet} \{(1, y\_1)\} \vdash 0$. We see that the results $\xrightarrow{11} \{(1, a)\} \vdash 0$ from the left and right-hand side are configuration congruent. For the remaining results, we see that the second left and right-hand side processes are equivalent in all but the fresh name in the registers.

The name of a bound name is unimportant in a process. The name only identifies its binding scope. When the bound name becomes free under fresh input/output, its name is placed in a register. For convenience, we use the bound name, as in this case. However, as we can see, the use of this causes difficulty for detecting congruence. As a result, we need to normalise the bound and fresh names.

Before the application of the transition rules, a configuration undergoes name normalisation steps. The n unique bound names in the process are converted to a normalised name of the form $\&1, \&2, ..., \&n$. So $a(x\_0).0 + a(y\_1).0$ becomes $a(\&1).0 + a(\&2).0$. We then apply the rules to obtain the fresh input results $\xrightarrow{11^\bullet} \{(1, \&1)\} \vdash 0$ for the left-hand

side and $\xrightarrow{11^\bullet}$ $\{(1, \&2)\} \vdash 0$ for the right-hand side. However, we are not finished yet because the names in the registers still differ.

Fresh name normalisation consists of scanning the register assignment and detecting names with the prefix '&'. For every name with the prefix '&', we generate the minimum available fresh name of the form #1, #2, ..., #n, $n \in \mathbb{N}$, and replace the name in the register with this generated fresh name. Also, the prefixed '&' names in the process are replaced with the corresponding prefixed '#' fresh name. When we apply this algorithm to the above results, we obtain $\{(1, \#1)\} \vdash 0$ and $\{(1, \#1)\} \vdash 0$. The configuration key of these two results are equivalent, which implies configuration congruence.

5.3.4 Nil process removal

There are three equations of structural congruence where the nil process appears, $\nu a.0 \equiv 0$, $P|0 \equiv 0$, and by rule of $P|Q \equiv Q|P$, $0|P \equiv 0$. Reducing the expression to their simplified counterparts is a matter of straightforward AST manipulation. A top-level function with signature

```
func rmNilProc(elem Element) Element
```

recursively traverses the AST and returns the next element. If a restriction is matched and its next element is a nil process, then return the nil process. The callee assigns this element as its next element, thus removing the restriction. Likewise, if a parallel is matched and its left-hand side process is an inaction, then return the right-hand side process. The procedure is the same for the right-hand side parallel process. If an element is neither a restriction nor a parallel, then assign its next/left/right element as the return element of the function call.

5.3.5 Unused restriction removal

From the structural congruence rules, we can derive the ability to remove unused restrictions.

$$
\begin{aligned}
\nu a.P &\equiv \nu a.(P|0) && P \equiv P|0 \\
&\equiv P|\nu a.0 \quad \text{if } a \notin \mathsf{fn}(P) && \nu a.(P|Q) \equiv P|\nu a.Q \quad \text{if } a \notin \mathsf{fn}(P) \\
&\equiv P|0 \quad \text{if } a \notin \mathsf{fn}(P) && \nu a.0 \equiv 0 \\
&\equiv P \quad \text{if } a \notin \mathsf{fn}(P) && P|0 \equiv P
\end{aligned}
$$

Therefore, if a process $\nu a.P$ does not use a restriction a, i.e., $a \notin \mathsf{fn}(P)$, then we can drop the restriction. In practice, this involves recursively traversing the AST to find restrictions and calling a helper function `appearsIn()` to return true or false whether a name is found in a process. If the name is used in the process, then the traversal function returns the restriction element and the callee assigns this as its next element. If the name is not used, then return the restriction's next element, thus removing the restriction from the AST.

5.3.6 Restriction scoping

The rule $\nu a.(P|Q) \equiv P|\nu a.Q$ if $a \notin \mathsf{fn}(P)$ narrows the scope of a restriction to the relevant process.

Example 5.2. Restriction scoping. Consider the processes

$$\nu a.\nu b.\nu c.(\bar{a}\langle a\rangle.0 \mid \bar{b}\langle b\rangle.\bar{c}\langle c\rangle.0)$$

and

$$\nu a.\bar{a}\langle a\rangle.0 \mid \nu b.\nu c.\bar{b}\langle b\rangle.\bar{c}\langle c\rangle.0$$

These two processes are structurally equivalent as per the rule above. The restrictions of the former expression can be scoped down to their respective processes. Restrictions preceding a parallel are found in the AST. In this case, νc is identified, which comes before a parallel. Using the `appearsIn()` function, we find whether the name c appears in the left-hand side process and right-hand side process. c does not appear in the LHS process but appears in the RHS process. With this result, we can place the restriction νc on the RHS process, resulting in $\nu a.\nu b.(\bar{a}\langle a\rangle.0 \mid \nu c.\bar{b}\langle b\rangle.\bar{c}\langle c\rangle.0)$. The results from `appearsIn()` and resulting behaviour can be encoded as a truth table.

restriction appears in LHS	*restriction appears in RHS*	*behaviour*
false	false	remove restriction
false	true	place restriction in RHS process
true	false	place restriction in LHS process
true	true	leave restriction

Tab. 5.1: Restriction preceding parallel scoping behaviour.

The algorithm operates bottom-up, so the same can be done for νb, for which we get $\nu a.(\bar{a}\langle a\rangle.0 \mid \nu b.\nu c.\bar{b}\langle b\rangle.\bar{c}\langle c\rangle.0)$. Finally, the procedure is applied for νa and this gives us the final result of $\nu a.\bar{a}\langle a\rangle.0 \mid \nu b.\nu c.\bar{b}\langle b\rangle.\bar{c}\langle c\rangle.0$, which is equivalent to the latter expression.

5.3.7 Restriction sorting

The rule $\nu a.\nu b.P \equiv \nu b.\nu a.P$ is made possible by adjusting the bound name normalisation algorithm before sorting the restrictions.

Example 5.3. Restriction sorting. Consider the two processes with an empty σ

$$\nu a.\nu b.a(x).b(y).0$$
$$\nu b.\nu a.a(x).b(y).0$$

From the rule, these two processes are equivalent. However, if we apply the name normalisation algorithm and replace the restriction bound names with the generated bound name, we get

$$\nu\&1.\nu\&2.\&1(\&3).\&2(\&4).0$$
$$\nu\&1.\nu\&2.\&2(\&3).\&1(\&4).0$$

which is not equal if we stringify the process. In order to resolve this, on the first pass of the bound name normalisation AST traversal, we only consider non-restriction elements. So after the first pass for each process, the result becomes

$$\nu a.\nu b.\&0(\&2).\&1(\&3).0$$
$$\nu b.\nu a.\&0(\&2).\&1(\&3).0$$

A map containing a pair $(oldName, newName)$ was stored. On the second pass, the restriction elements are renamed based on the map pairs, resulting in

$$\nu\&0.\nu\&1.\&0(\&2).\&1(\&3).0$$
$$\nu\&1.\nu\&0.\&0(\&2).\&1(\&3).0$$

Finally, we sort the restrictions lexicographically by gathering adjacent restrictions and building a sorted linked list of restriction elements. The processes become

$$\nu\&0.\nu\&1.\&0(\&2).\&1(\&3).0$$
$$\nu\&0.\nu\&1.\&0(\&2).\&1(\&3).0$$

which are equal. Since both of the registers are empty, these two configurations are configuration congruent.

5.3.8 Summation and composition sorting

This algorithm implements structural congruence rule (2), *Reordering of terms in a summation*, and the remaining equations in rule (3), $P|Q \equiv Q|P$ and $P|(Q|R) \equiv (P|Q)|R$. Summation and composition expressions with different bracketing and process positioning are converted into a standard form to detect for structural congruence.

Example 5.4. Summation and composition sorting. Consider the two expressions (1) and (2)

$$(P \mid Q) \mid ((A + C) + B) \qquad (1)$$
$$((B + (C + A)) \mid Q) \mid P \qquad (2)$$

From the rules, these two expressions are structurally congruent. When we normalise these expressions and stringify the process, we obtain equivalent strings. We define a top-level function to normalise a process to be

```
func sortSumPar(elem Element) Element
```

which follows the same procedure as `rmNilProc()` where the AST is traversed recursively and returns the next element. When a parallel element is encountered, the non-parallel children of the adjacent parallels are recursively gathered. For process (1), the elements collected are $\{P, Q, (A + C) + B\}$. A call is made to `sortSumPar()` for each of these elements. In this manner, the summations and compositions are sorted bottom-up.

We call `sortSumPar()` with the element $(A + C) + B$. The procedure is identical for summation. The elements collected are $\{A, C, B\}$. As all of these elements contain no sum/parallels, calling the top-level function yields no effect. We sort the element list $\{A, C, B\}$ based on the lexicographic order of the stringified process and get $\{A, B, C\}$. The next step is building a right-leaning summation tree from these sorted elements, which gives $A + (B + C)$. Finally, we return this tree.

Returning from the `sortSumPar()` call, the element $(A + C) + B$ becomes $A + (B + C)$. When we sort the list, as above, we get $\{A + (B + C), P, Q\}$. Again, a right-leaning tree is constructed with these sorted elements, this time with compositions. We get the normalised form $(A + (B + C)) \mid (P \mid Q)$ as the final result.

When we apply the algorithm above for process (2), we expect to receive the same normalised form.

5.3.9 Process definition

Structural congruence rule (5) defining process definitions $A\langle\vec{b}\rangle \equiv \{\vec{b}/\vec{a}\}P_A$ if $A(\vec{a}) \stackrel{\text{def}}{=} P_A$ is implicitly implemented by the transition rule Rec. The Rec rule unfolds the process definitions, substitutes the names and takes the process definition's next transitions.

Example 5.5. Process definition congruence. Consider the configuration and process definition P

$$\{(1, a)\} \vdash P + \bar{a}\langle a\rangle.0 \quad \textit{with definition} \quad P = \bar{a}\langle a\rangle.0$$

This configuration only has one transition $\xrightarrow{\bar{1}1}$ $\{(1, a)\} \vdash 0$, regardless of whichever side of the summation process it takes. This is because the process on the left and right-hand side are equivalent. This also demonstrates that the Res rule already captures the structural congruence process definition rule.

5.4 Garbage collection

A further approach to reduce the amount of potentially equivalent states is to garbage collect the unused names in the registers.

Lemma 5.3. Garbarge collection (GC). Given the configuration $\sigma \vdash P$, if

$$\sigma' = \{(i, a) \mid \sigma(i) = a, a \in \mathsf{fn}(P)\}$$

then $\sigma \vdash P \overset{\times\pi}{\sim} \sigma' \vdash P$, where $\overset{\times\pi}{\sim}$ is an $\times\pi$-bisimulation.

Let σ' be the garbage collected register assignment such that for all the names in the registers σ', there exists a free name in P corresponding to that name. In other words, all registers with names $a \notin \mathsf{fn}(P)$ are removed from the register assignment σ. For example, the garbage collected configuration of $\{(1, a), (2, b), (3, c), (4, d)\} \vdash \bar{b}\langle d\rangle.0$ is $\{(2, b), (4, d)\} \vdash \bar{b}\langle d\rangle.0$ because a and c are not used in P. The assumption is that $\sigma \vdash P$ and $\sigma' \vdash P$ are $\times\pi$-bisimilar.

It may seem odd that the configurations of the original and filtered registers are bisimiliar. For example, Inp1's transitions depends on the contents of the registers. However, Tzevelekos' definition of bisimilarity is not in the standard sense. $\times\pi$-bisimulation is based on a notion of register matching, which makes this equivalence possible. This lemma has been verified by Tzevelekos. Since the GC'ed configuration and non-GC'ed configuration are bisimiliar, garbage collection can be implemented.

Labelled Transition System 6

The application of congruence rules transforms the configurations into a normal form, which allows us to detect equivalent configurations which appear seemingly different. With this, we can generate states, apply these rules, and explore the states of a model, thus generating the labelled transition system.

6.1 LTS generation structure

We define structs to represent the LTS.

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

type Transition struct {
    Source      int
    Destination int
    Label       Label
}
```

Listing 6.1: Data types to represent the labelled transition system.

`Lts` is a directed graph of the labelled transition system, consisting of a set of states and list of transitions. The set of states are represented by a map which maps a state ID to a configuration. The `Transition` data type comprises of the source and destination state ID and transition label.

6.2 State exploration

The LTS is explored in a breadth-first search (BFS) manner. The psuedocode is provided below and is supplemented with an example.

Algorithm 6.1 Explore the states and return the labelled transition system

```
1:  function EXPLORE(root) return LTS{states, trns}
2:      visited ← {}, trnsSeen ← {}, states ← {}, trns ← {}
3:      APPLYCONGRUENCE(root)
4:      key ← GETCONFKEY(root)
5:      visited[key] ← 0, states[0] ← root
6:      stateID ← 1, queue ← queue().push(root)
7:      statesExplored ← 0
8:      for queue.length > 0 ∧ statesExplored < maxStatesExplored do
9:          src ← queue.dequeue()
10:         srcID ← visited[GETCONFKEY(src)]
11:         confs ← TRANS(src)
12:         for conf ∈ confs do
13:             APPLYCONGRUENCE(conf)
14:             dstKey ← GETCONFKEY(conf)
15:             if dstKey ∉ visited then
16:                 visited[dstKey] ← stateID
17:                 states[stateID] ← conf
18:                 stateID++
19:                 queue.push(conf)
20:             trn ← TRANSITION{srcID, visited[dstKey], conf.Label}
21:             if trn ∉ trnsSeen then
22:                 trnsSeen ← trnsSeen ∪ trn
23:                 trns ← trns ∪ trn
24:         statesExplored++
```

Example 6.1. State exploration. Consider a number of states defined by the set of final configurations $O(K)$,

$$s_0, s_1, s_2, s_3 \in O(K)$$

The root configuration is always s_0. The root configuration undergoes structural and configuration congruence rules. The key of the configuration is obtained in the form of a stringified register assignment and process. This key is the unique identifier of the state, and is used to detect previously visited states of the same congruence.

The LTS is traversed breadth-first using a queue. We pass s_0 to `trans()` and get the next states s_1, s_2, and s_3. For s_1, we apply congruence rules and garbage collection, and obtain the configuration's key. This state has not been encountered before, so we append s_1 to the queue, as well as to the transition results list. This repeats for s_2. However, for s_3, we see that that the configuration's key matches s_1's key and the transition $s_0 \xrightarrow{\alpha} s_1 \equiv s_0 \xrightarrow{\alpha} s_3$ has been encountered before. In this case, we do not append neither s_3 to the queue, nor its transition to the results list.

The BFS queue is dequeued and the procedure repeats until there are no states left in the queue or if the number of states explored reaches its maximum user-specified limit (the default is 20).

6.3 LTS output

After exploring the states, we are left with a labelled transition system data structure which we can output to any form of our choosing. We output the LTS in two formats – a pretty-printed form and a graph description language.

6.3.1 Pretty-printed LTS

Given the expression

```
$x.a'<x>.b'<x>.0 | b(y).0
```

the pretty-printed LTS output is

```
s0 = {(1,#1),(2,#2)} |- (#2(&2).0 | $&1.#1'<&1>.#2'<&1>.0)
s0  2 1   s1 = {(1,#1),(2,#2)} |- $&1.#1'<&1>.#2'<&1>.0
s0  2 2   s1 = {(1,#1),(2,#2)} |- $&1.#1'<&1>.#2'<&1>.0
s0  2 3*  s1 = {(1,#1),(2,#2)} |- $&1.#1'<&1>.#2'<&1>.0
s0  1'1^  s2 = {(1,#1),(2,#2)} |- (#2'<#1>.0 | #2(&1).0)
s1  1'1^  s3 = {(1,#1),(2,#2)} |- #2'<#1>.0
s2  2'1   s4 = {(2,#2)} |- #2(&1).0
s2  2 1   s3 = {(1,#1),(2,#2)} |- #2'<#1>.0
s2  2 2   s3 = {(1,#1),(2,#2)} |- #2'<#1>.0
s2  2 3*  s3 = {(1,#1),(2,#2)} |- #2'<#1>.0
s2  t     s5 = {} |- 0
s3  2'1   s5 = {} |- 0
s4  2 2   s5 = {} |- 0
s4  2 1*  s5 = {} |- 0
```

Listing 6.2: Pretty-printed LTS output.

The LTS is outputted as human-readable ASCII text for use in inspecting the results in the command-line interface or in a text file. The symbols used match closely to the mathematical syntax, as to be as intuitive to the user as possible. There is some use of special characters, which are denoted below.

pretty-printed notation	*mathematical notation*	
`{} |- P`	$\{\} \vdash P$	configuration
`#1`	a_1	free name
`&1`	x_1	bound name
`1 1`	$1\,1$	known input
`1'1`	$\bar{1}\,1$	known output
`1 1*`	$1\,1^{\bullet}$	fresh input
`1'1^`	$\bar{1}\,1^{\circledast}$	fresh output
`t`	τ	tau step

Tab. 6.1: Pretty-printed LTS output legend.

The line `s0 2 3* s1` means that from state s_0 take the transition $2\,3^{\bullet}$ to reach s_1. s_1's configuration is shown after the `=`.

6.3.2 Graph description language

The graph description language used is the *GraphViz DOT* language [26]. *GraphViz* is an open-source tool which generates visualisations of graphs defined by the *DOT* language. In addition, many libraries in various programming languages exist for parsing the *DOT* language, should one wish to import the LTS file.

A template was written which captures the syntax of the *DOT* language and generates a file based on the LTS provided. So for the same program as above, we get the following file.

```
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="t"]
    s3 -> s5 [label="2' 1"]
    s4 -> s5 [label="2 2"]
    s4 -> s5 [label="2 1*"]
}
```

Listing 6.3: LTS output in the *GraphViz DOT* language (UTF-8 characters are replaced with ASCII equivalents).

Using the *GraphViz* toolset, we generate the graph visualisation with the command

```
$ dot -Tpdf -o lts.pdf lts.dot
```

and obtain the following graphical visualisation.

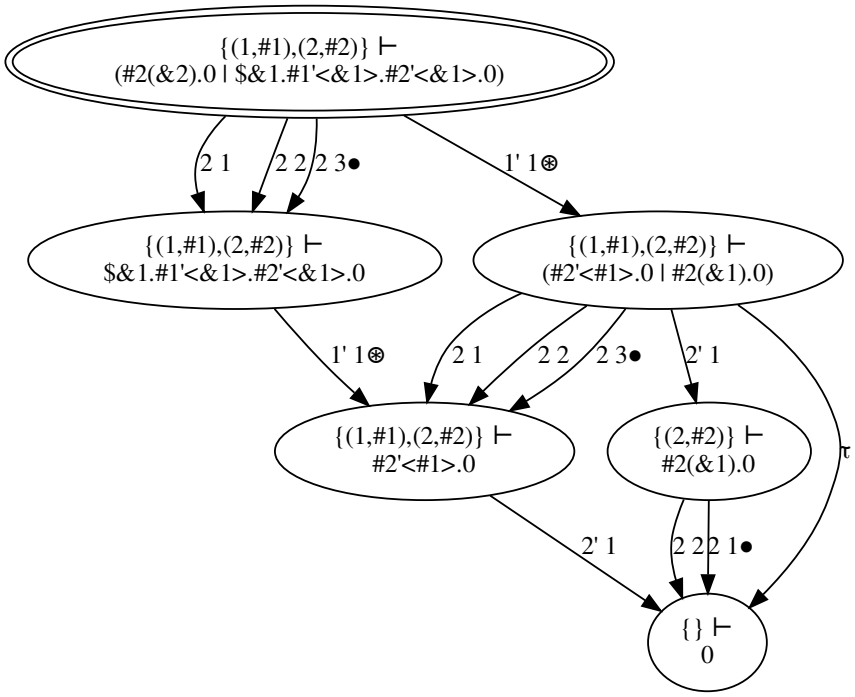


Fig. 6.1: LTS visualisation from the *GraphViz DOT* language with configurations as nodes.

Note that the starting state is denoted by a double periphery. The user can also specify the use of state numbers as nodes instead of configurations, which is helpful for cases when the graph becomes too large.

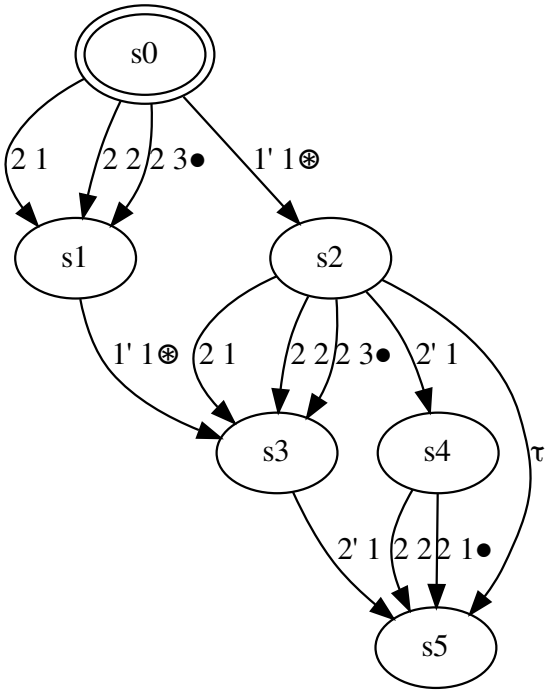


Fig. 6.2: LTS visualisation from the *GraphViz DOT* language with state numbers as nodes.

The LTS can also be outputted as a *DOT* file which uses *LaTeX* labels. This allows for the use of standard mathematical notation for configurations and is useful for cases where the user would not be too familiar with the meanings of the special symbols. Using the *dot2tex* tool [27], the *DOT* file can be converted to a *LaTeX* file, which can then be compiled into a *PDF* containing the visualisation. The commands below

```
$ dot2tex -o lts.tex lts.dot
$ pdflatex lts.tex
```

generate the following visualisation.

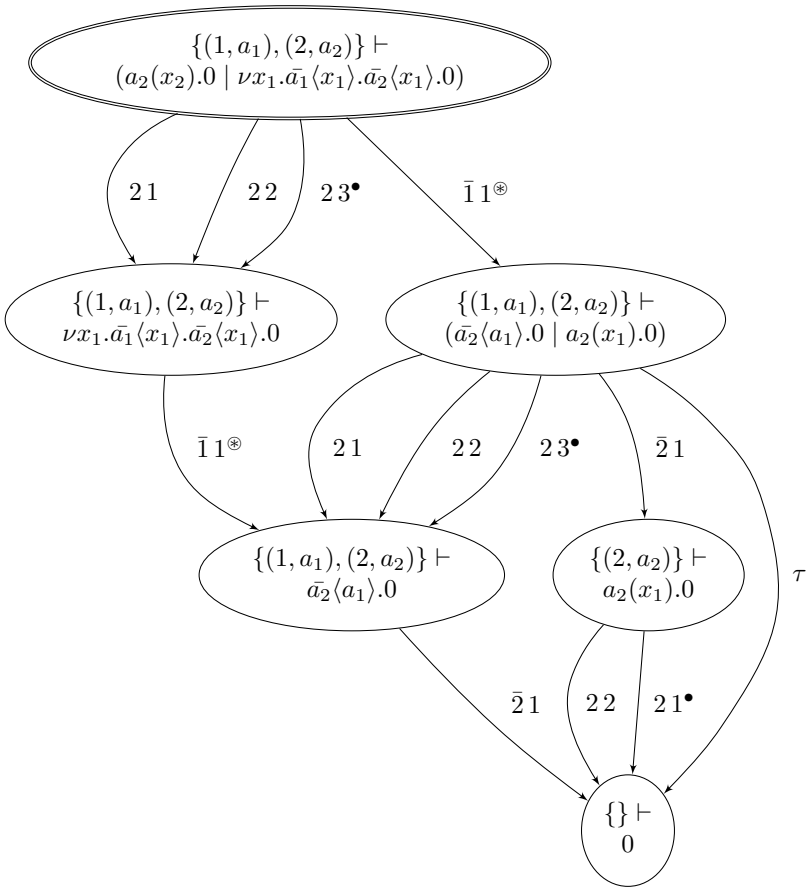


Fig. 6.3: LTS visualisation with configurations produced using *dot2tex*.

The free names are named as a subscripted with their number and the bound names are named as x, also subscripted with their number.

We can also specify the use of state numbers instead of configurations with *LaTeX* styling.

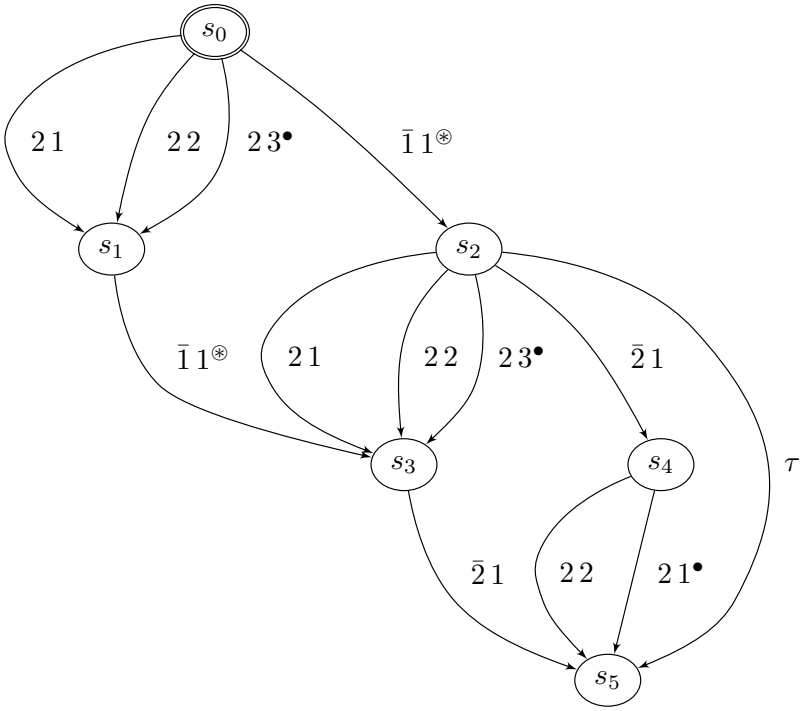


Fig. 6.4: LTS visualisation with state numbers produced using *dot2tex*.

6.4 Command-line tool

The tool is built as a command-line program. Its name is `pifra`, an acronym for the pi-calculus fresh-register automata. Command-line flags are used to adjust program parameters and output styles. The full range of options are presented below.

```
$ pifra --help
pifra generates labelled transition systems (LTS) of
pi-calculus models represented by fresh-register automata.

Usage:
pifra [OPTION...] FILE

Options:
  -n, --max-states int         maximum number of states explored (default 20)
  -r, --max-registers int      maximum number of registers (default is unlimited)
  -d, --disable-gc             disable garbage collection
  -i, --interactive            inspect interactively the LTS in a prompt
  -o, --output string          output the LTS to a file (default format is the Graphviz DOT language)
  -t, --output-tex             output the LTS file with LaTeX labels for use with dot2tex
  -p, --output-pretty          output the LTS file in a pretty-printed format
  -s, --output-states          output state numbers instead of configurations for the Graphviz DOT file
  -l, --output-layout string   layout of the GraphViz DOT file, e.g., "rankdir=TB; margin=0;"
  -q, --quiet                  do not print or output the LTS
  -v, --stats                  print LTS generation statistics
  -h, --help                   show this help message and exit
```

Listing 6.4: Help message of the `pifra` program.

7 Verification

With the generation of the LTS, we open up the possibility of performing some simple model checking verification. We are able to prove or disprove if a system reaches a certain state.

7.1 Name marking

The ability to 'mark' names is added to the tool to assess for state reachability.

Example 7.1. Name marking. Consider the program

```
$a.a'<a>.b'<b>.c'<c>.0
```

This model only has one state regardless of the context, namely

```
s0 = {(1,#1),(2,#2)} |- $&1.&1'<&1>.#1'<#1>.#2'<#2>.0
```

This is because the name a is restricted and we attempt to send a name on this restricted channel, which will never have any receivers. As a result, the proceeding output action $\bar{b}\langle b\rangle$ is impossible. The names in the configuration are normalised after the application of congruence rules, hence `&1` for a, `#1` for b, and `#2` for c. Note that in this simple case, b will always reside in the register label 1. Therefore, if we check that the transition $\bar{1}1$ (known output) does not exist in the LTS, then the model cannot reach this state, i.e., running

```
$ pifra lts.pi | grep "1'1"
```

yields no results, and verifies that this state cannot be reached. After name normalisation, it is difficult to keep track of which name appears where, especially if the names in the registers change. Hence, it would be helpful if our target name could be marked. Names can be marked with the prefix `"_"`. So the above program marking the name b becomes

```
$a.a'<a>._b'<_b>.c'<c>.0
```

and its only state becomes

```
s0 = {(1,_b),(2,#1)} |- $&1.&1'<&1>._b'<_b>.#1'<#1>.0
```

Marked names are consistently placed at the starting labels of the register assignment of the root configuration. For cases of non-state reachability, it is presumed that the marked name will persist throughout all states, hence avoiding garbage collection.

7.2 State reachability

To summarise the above example, state reachability verification can be performed by the following. In the pi-calculus model of a system,

- designate an output `_b'<_b>` to signal the target state, marking the name as `_b`,
- generate the LTS, and
- search the output for the transition `1'1`.

If the transition `1'1` exists, then this state has been reached. If not, then this state is not reachable in the model (if exhaustively explored, i.e., its LTS is finite). As a result, we can mark any target state and check if it is reachable or not.

7.3 Verification: Examples

Example 7.2. Fresh name generation. This example was provided by Koutavas. Consider the two systems

```
GenFreshA(a) = $fr. a'<fr>. GenFreshA(a)
```

and

```
GenFreshB(a) = $fr5. GF1(a,fr5)
GF1(a,fr5) = $fr1. a'<fr1>. GF2(a,fr5)
GF2(a,fr5) = $fr2. a'<fr2>. GF3(a,fr5)
GF3(a,fr5) = $fr3. a'<fr3>. GF4(a,fr5)
GF4(a,fr5) = $fr4. a'<fr4>. GF5(a,fr5)
GF5(a,fr5) =       a'<fr5>. GF1(a,fr5) + $fr6. a'<fr6>. GF1(a,fr5)
```

These two systems are not bisimilar and are not equivalent under any reasonable equivalence relation. The reason is that the former generates an infinite stream of fresh names on `a`, whereas the latter may reuse the 5^{th} output. Note that for `GF5`, there are two options. The first choice outputs the bound name `fr5` as a fresh name, and may use this name to output again after 5 outputs. The other choice outputs a fresh name, and repeats the cycle. Therefore, we cannot guarantee that the latter system outputs a fresh name every time, in contrast with the former system. With the current state of the tool, there is no manner to formally prove or disprove bisimilarity. However, we can create a test which outputs on a marked name representing the target state if it detects the same fresh output twice. The test is

```
Test(a) = Inp(a) | a(x). a(y). [x=y] _BAD'<_BAD>. $dummy. dummy'<_BAD>.0
Inp(a) = a(z). Inp(a)
```

An input process `Inp` performs arbitrary inputs to capture any potential non-consecutive identical outputs (as in the case of the latter system). The second process of `Test` inputs two names and checks if they are equal. If they are, then our target state `_BAD'<_BAD>` would be reached. For clarity, the `dummy` action proceeding it preserves the name `_BAD` in the register assignment so garbage collection will not collect it. So when searching for this bad state, its transition label name is clear.

We can now consider two tests

```
$a. (GenFreshA(a) | Test(a))
```

and

```
$a. (GenFreshB(a) | Test(a))
```

which consist of their respective systems in parallel with the test process and restricts the channel a. This binding of a is required to restrict the behaviour to only the above system. If a is not bound, we may receive two identical names on a from the context, thus defeating the purpose of the system test.

We generate the respective exhaustive LTS's (both of which are finite-state models) and search for the action transition 1'1.

```
$ pifra -n 100 gen-fresh-a.pi | grep "1'1"
$ pifra -n 100 gen-fresh-b.pi | grep "1'1"
s28  1'1   s31 = {(1,_BAD)} |- ($&3.&3'<_BAD>.0 | $&1.$&2.(GF1(&1, &2) | Inp(&1)))
```

The LTS of GenFreshA does not have this action, meaning that the 'bad' state was never reached. The LTS of GenFreshB finds an instance of the bad state being reached. This verifies the property that an infinite stream of fresh outputs occurs for the first system, and disproves it for the second system.

This verification procedure of encoding target states in the system and grep-ing for these states is performed manually. For different systems and different properties, we would have different tests. If the LTS is finite, we can exhaustively check for all states and prove whether the system satisfies the property. However, for infinite-state LTS's, we can only prove that the system satisfies the property up to the given number of states explored. With this, it is better to encode a certain property in an infinite-state LTS and attempt to disprove it, rather than to prove it.

Example 7.3. Password system. We examine an example devised by Koutavas. Consider a system where a secret is generated from a password.

```
GenPass(requestNewPass) = requestNewPass(x). $pass. x'<pass>.0

KeepSecret(requestNewPass) = $p. requestNewPass'<p>. p(pass). (
    StoreSecret(pass) | TestSecret(pass) )

StoreSecret(pass) = $secret. pass'<secret>. StoreSecret(pass)

TestSecret(pass) = pub(x). pass(secret). (
    TestSecret(pass) + [x=secret] _BAD'<_BAD>.0 )

$requestNewPass. (
    GenPass(requestNewPass)  |  KeepSecret(requestNewPass) )
```

The system is initialised by the process KeepSecret, which sends a restricted channel p over to the GenPass. GenPass generates a password pass and sends this over to KeepSecret on the restricted channel. KeepSecret now does two things – it continuously generates a

secret and sends this over the password channel in `StoreSecret` and checks to see if this secret can appear over the public channel `pub` in `TestSecret`.

We evaluate whether the secret is exposed by checking if the input on the public channel is ever equal to the secret. If this occurs, the target state `_BAD'<_BAD>` would be reached by action `1'1`. We check for this in the finite-state model with

```
$ pifra password.pi | grep "1'1"
```

and find no results. Therefore, the system is secure. The assertion `x=secret` will never be true, no matter how the context of this system interacts with it. We can conclude that the context will never learn the secret.

We investigate a scenario which makes this system insecure. Suppose that the channel `requestNewPass` is not restricted, i.e., the system is initialised with

```
GenPass(requestNewPass)  |  KeepSecret(requestNewPass)
```

We grep for our target state and find

```
$ pifra -n 127 password-insecure.pi | grep "1'1"
s126  1'1   s651 = {(1,_BAD),(3,#2)} |- (GenPass(#2) | StoreSecret(_BAD))
```

which proves that this system is not secure, as the target state was reached. This is because the communication channel between `KeepSecret` and `GenPass` is not restricted. Think about these two processes as computers over a network. A third process, called `Attacker` could intercept the channel if `requestNewPass` is not secure.

```
Attacker(requestNewPass) =
    requestNewPass(x). $pass. x'<pass>. pass(secret). pub'<secret>. 0
```

The rogue process generates a password and sends it over to `KeepSecret`, as with `GenPass`. Additionally, the process gathers the secret from the password channel and sends over the secret to the public channel `pub`, which triggers the assertion. This behaviour is captured by the LTS without explicitly encoding the attacker. The LTS accounts for all behaviour of the model when placed in any context, and thus finds that these set of insecure actions are a possibility.

Example 7.4. Intermediate server system. We revisit the example described in the motivating example of the introduction. Recall that a message `"hello"` is sent from client A to client B through an intermediate server S. This system is modelled in the pi-calculus as the following.

```
A(as)    = $ab. as'<ab>. ab'<hello>. 0

B(sb)    = sb(chnl). chnl(msg). [msg!=hello] _BAD'<_BAD>. 0

S(as,sb) = as(chnl). sb'<chnl>. 0

$as. $sb. ( A(as) | B(sb) | S(as, sb) )
```

We also encode the safety property that the message `msg` received on B must be `"hello"`. To check for this, if the message is not `"hello"`, then the target state `_BAD'<_BAD>` would be reached, signalling a violation of our property. We produce the LTS

```
$ pifra server.pi
s0 = {(1,_BAD),(2,#1)} |- $&1.($&2.(B(&2) | S(&1, &2)) | A(&1))
s0  t     s1 = {(1,_BAD),(2,#1)} |- $&2.($&1.(&1'<&2>.0 | B(&1)) | &2'<#1>.0)
s1  t     s2 = {(1,_BAD),(2,#1)} |- $&1.(&1'<#1>.0 | &1(&2).[&2!=#1]_BAD'<_BAD>.0)
s2  t     s3 = {(1,_BAD),(2,#1)} |- [#1!=#1]_BAD'<_BAD>.0
```

and find that the action `1'1` does not appear, thus verifying the property. From the LTS, we can see clearly the behaviour of the system. Given the initial state, the only transition is a number of tau steps to reach a final state where the name `msg` will always be `"hello"`.

Consider the case where the name `as` is not restricted, i.e., we initialise the system with

```
$sb. ( A(as) | B(sb) | S(as, sb) )
```

In this case, we expose `as` to the context, which may lead to the scenario where a client C sends a different message over to B, as in the introductory example. We check for this with

```
$ pifra -n 22 server2.pi | grep "1'1"
s21  1'1   s42 = {(2,#1),(3,#2)} |- A(#1)
```

and it results in the violation of the property. Likewise, suppose that `sb` is not restricted, i.e., we initialise the system with

```
$as. ( A(as) | B(sb) | S(as, sb) )
```

We can reason why the message received on B would not be guaranteed to be `"hello"`. A channel could be sent to `sb` and a different message could be sent over this channel. We check this with

```
$ pifra -n 7 server3.pi | grep "1'1"
s6  1'1   s23 = {(2,#1),(3,#2)} |- $&1.(A(&1) | S(&1, #2))
```

which confirms our reasoning, and verifies the violation of the property. With certain names unrestricted, we can see how subtle changes in the system can result in unintended interactions with the context. By generating the LTS, we can detect these behaviours from seemingly correct systems. This is important when placing a small part into a larger system. We want to ensure that the small part does not result in unforeseen behaviour.

The above are examples of how a model could be verified for certain properties. All safety properties can be encoded using runtime monitors [28, p. 34]. Monitors work by monitoring execution steps of some system and checks if it violates a security policy by reaching a target state. This is what we do when we use `_BAD'<_BAD>` to signal the target state. Schneider [28, p. 31] refers to this as *execution monitoring* (EM) which would be applicable to security in kernels, firewalls, and operating systems.

Evaluation 8

The program is evaluated with finite-state and infinite-state models using certain criteria for each.

8.1 Finite-state models

For each finite-state model, we investigate the LTS output and the LTS generation statistics.

Tzevelekos' example

This model is an example from the FRA paper [5, p. 303, example 31], and is referred to in the global freshness example 2.14.

```
P(a,b) = a'<b>.$c.P(b,c)
$b.P(a,b)
```

Listing 8.1: *tzevelekos* model.

The program gives the LTS as described in the example. Its pretty-printed LTS is

```
s0 = {(1,#1)} |- $&1.P(#1, &1)
s0  1'1^  s0 = {(1,#1)} |- $&1.P(#1, &1)
```

Listing 8.2: *tzevelekos* LTS.

and its graph visualisation generated from the corresponding *Graphviz DOT* file with *LaTeX* labels is

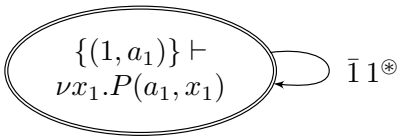


Fig. 8.1: *tzevelekos* visualisation.

The LTS generation statistics are

```
states explored      1
states generated     1
states unique        1
transitions          1
time I/O             302.5µs
time LTS generation  350.2µs
```

Listing 8.3: *tzevelekos* statistics.

and each statistic is described as the following.

- A state which is explored means that `trans()` was called using that state, and that all transitions from this state have been exhaustively found.
- The number of generated states is the total number of states obtained by calling `trans()` from an explored state.
- The number of unique states are those that have been uniquely identified after undergoing congruence rules. This is also the number of states shown in the LTS.
- The number of transitions is the total number of labelled actions in the LTS.
- The input/output (I/O) time is the total wall clock time spent reading the input file and writing/printing the LTS output.
- The LTS generation time is the total wall clock time spent generating the LTS, which includes parsing the model and gathering the states and transitions of the LTS.

vk-fin-st1

The models prefixed by *'vk'* were written by Koutavas, who provided a test suite to evaluate the program. *'fin-st'* means finite-state.

```
P = a(x).($y.x'<y>.P)
P
```

Listing 8.4: *vk-fin-st1* model.

```
s0 = {(1,#1)} |- P
s0  1 1   s1 = {(1,#1)} |- $&1.#1'<&1>.P
s0  1 2*  s2 = {(1,#1),(2,#2)} |- $&1.#2'<&1>.P
s1  1'2^  s0 = {(1,#1)} |- P
s2  2'2^  s0 = {(1,#1)} |- P
```

Listing 8.5: *vk-fin-st1* LTS.

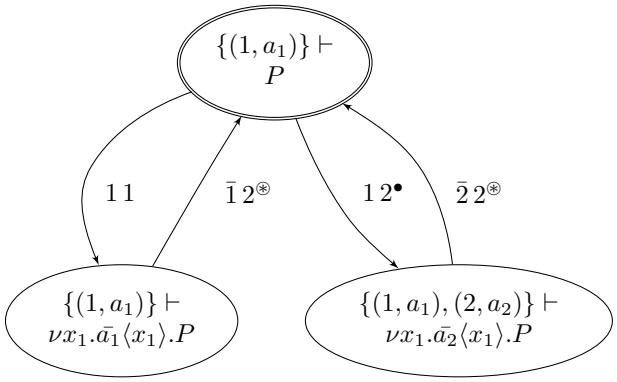


Fig. 8.2: *vk-fin-st1* visualisation.

```
states explored      3
states generated     4
states unique        3
transitions          4
time I/O             566.9µs
time LTS generation  832.3µs
```

Listing 8.6: *vk-fin-st1* statistics.

The number of generated states is 4 and the number of unique states is 3, meaning that one state was identified to be congruent. For exhaustively explored finite-state models, the explored states is equal to the unique states, as we explored every possible state in the LTS.

The remainder of the finite-state models are left for the reader to inspect.

vk-fin-st2

```
P(a) = a(x).$y.(x'<y>.0 | y(z).P(z))
P(a)
```

Listing 8.7: *vk-fin-st2* model.

```
s0 = {(1,#1)} |- P(#1)
s0  1 1   s1 = {(1,#1)} |- $&1.(#1'<&1>.0 | &1(&2).P(&2))
s0  1 1*  s1 = {(1,#1)} |- $&1.(#1'<&1>.0 | &1(&2).P(&2))
s1  1'1^  s2 = {(1,#1)} |- #1(&1).P(&1)
s2  1 1   s0 = {(1,#1)} |- P(#1)
s2  1 1*  s0 = {(1,#1)} |- P(#1)
```

Listing 8.8: *vk-fin-st2* LTS.

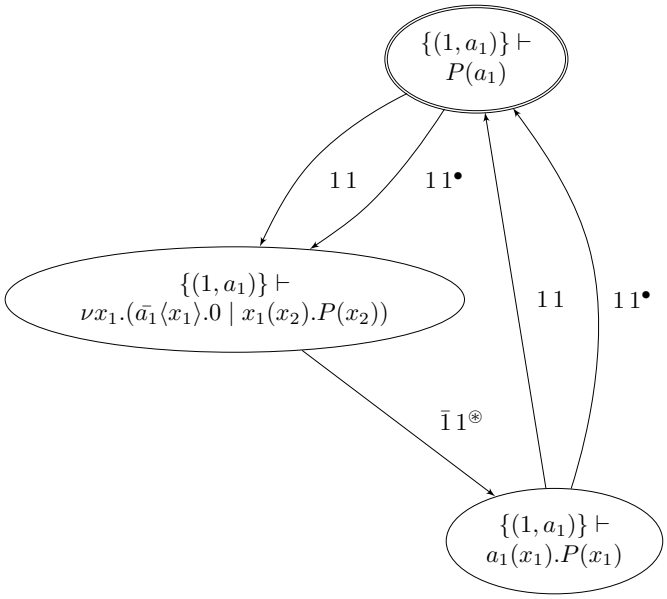


Fig. 8.3: *vk-fin-st2* visualisation.

```
states explored      3
states generated     5
states unique        3
transitions          5
time I/O             368.2µs
time LTS generation  839.3µs
```

Listing 8.9: *vk-fin-st2* statistics.

vk-fin-st3

```
P(a) = a(x).$y.(x'<y>.0 | b(z).[z=y] P(a))
P(a)
```

Listing 8.10: *vk-fin-st3* model.

```
s0 = {(1,#1),(2,#2)} |- P(#1)
s0  1 1   s1 = {(1,#1),(2,#2)} |- $&1.(#1'<&1>.0 | #2(&2).[&2=&1]P(#1))
s0  1 2   s2 = {(1,#1),(2,#2)} |- $&1.(#2'<&1>.0 | #2(&2).[&2=&1]P(#1))
s0  1 3*  s3 = {(1,#1),(2,#2),(3,#3)} |- $&1.(#2(&2).[&2=&1]P(#1) | #3'<&1>.0)
s1  1'3^  s4 = {(1,#1),(2,#2),(3,#3)} |- #2(&1).[&1=#3]P(#1)
s1  2 1   s5 = {(1,#1),(2,#2)} |- $&1.(#1'<&1>.0 | [#1=&1]P(#1))
s1  2 2   s6 = {(1,#1),(2,#2)} |- $&1.(#1'<&1>.0 | [#2=&1]P(#1))
s1  2 3*  s7 = {(1,#1),(2,#2),(3,#3)} |- $&1.(#1'<&1>.0 | [#3=&1]P(#1))
s2  2'3^  s4 = {(1,#1),(2,#2),(3,#3)} |- #2(&1).[&1=#3]P(#1)
s2  2 1   s8 = {(1,#1),(2,#2)} |- $&1.(#2'<&1>.0 | [#1=&1]P(#1))
s2  2 2   s9 = {(1,#1),(2,#2)} |- $&1.(#2'<&1>.0 | [#2=&1]P(#1))
s2  2 3*  s10 = {(1,#1),(2,#2),(3,#3)} |- $&1.(#2'<&1>.0 | [#3=&1]P(#1))
s2  t     s11 = {(1,#1),(2,#2)} |- $&1.[&1=&1]P(#1)
s3  2 1   s12 = {(1,#1),(2,#2),(3,#3)} |- $&1.(#3'<&1>.0 | [#1=&1]P(#1))
s3  2 2   s13 = {(1,#1),(2,#2),(3,#3)} |- $&1.(#3'<&1>.0 | [#2=&1]P(#1))
s3  2 3   s14 = {(1,#1),(2,#2),(3,#3)} |- $&1.(#3'<&1>.0 | [#3=&1]P(#1))
s3  2 4*  s15 = {(1,#1),(2,#2),(3,#3),(4,#4)} |- $&1.(#3'<&1>.0 | [#4=&1]P(#1))
s3  3'3^  s4 = {(1,#1),(2,#2),(3,#3)} |- #2(&1).[&1=#3]P(#1)
s4  2 1   s16 = {(1,#1),(2,#2),(3,#3)} |- [#1=#3]P(#1)
s4  2 2   s17 = {(1,#1),(2,#2),(3,#3)} |- [#2=#3]P(#1)
s4  2 3   s18 = {(1,#1),(2,#2),(3,#3)} |- [#3=#3]P(#1)
s4  2 4*  s19 = {(1,#1),(2,#2),(3,#3),(4,#4)} |- [#4=#3]P(#1)
s5  1'3^  s16 = {(1,#1),(2,#2),(3,#3)} |- [#1=#3]P(#1)
s6  1'3^  s17 = {(1,#1),(2,#2),(3,#3)} |- [#2=#3]P(#1)
s7  1'4^  s20 = {(1,#1),(2,#2),(3,#3),(4,#4)} |- [#3=#4]P(#1)
s8  2'3^  s16 = {(1,#1),(2,#2),(3,#3)} |- [#1=#3]P(#1)
s9  2'3^  s17 = {(1,#1),(2,#2),(3,#3)} |- [#2=#3]P(#1)
s10  2'4^  s20 = {(1,#1),(2,#2),(3,#3),(4,#4)} |- [#3=#4]P(#1)
s11  1 1   s1 = {(1,#1),(2,#2)} |- $&1.(#1'<&1>.0 | #2(&2).[&2=&1]P(#1))
s11  1 2   s2 = {(1,#1),(2,#2)} |- $&1.(#2'<&1>.0 | #2(&2).[&2=&1]P(#1))
s11  1 3*  s3 = {(1,#1),(2,#2),(3,#3)} |- $&1.(#2(&2).[&2=&1]P(#1) | #3'<&1>.0)
s12  3'3^  s16 = {(1,#1),(2,#2),(3,#3)} |- [#1=#3]P(#1)
s13  3'3^  s17 = {(1,#1),(2,#2),(3,#3)} |- [#2=#3]P(#1)
s14  3'4^  s20 = {(1,#1),(2,#2),(3,#3),(4,#4)} |- [#3=#4]P(#1)
s15  3'3^  s19 = {(1,#1),(2,#2),(3,#3),(4,#4)} |- [#4=#3]P(#1)
s18  1 1   s1 = {(1,#1),(2,#2)} |- $&1.(#1'<&1>.0 | #2(&2).[&2=&1]P(#1))
s18  1 2   s2 = {(1,#1),(2,#2)} |- $&1.(#2'<&1>.0 | #2(&2).[&2=&1]P(#1))
s18  1 3   s3 = {(1,#1),(2,#2),(3,#3)} |- $&1.(#2(&2).[&2=&1]P(#1) | #3'<&1>.0)
s18  1 3*  s3 = {(1,#1),(2,#2),(3,#3)} |- $&1.(#2(&2).[&2=&1]P(#1) | #3'<&1>.0)
```

Listing 8.11: *vk-fin-st3* LTS.

```
states explored      21
states generated     38
states unique        21
transitions          38
time I/O             338.3µs
time LTS generation  3.6937ms
```

Listing 8.12: *vk-fin-st3* statistics.

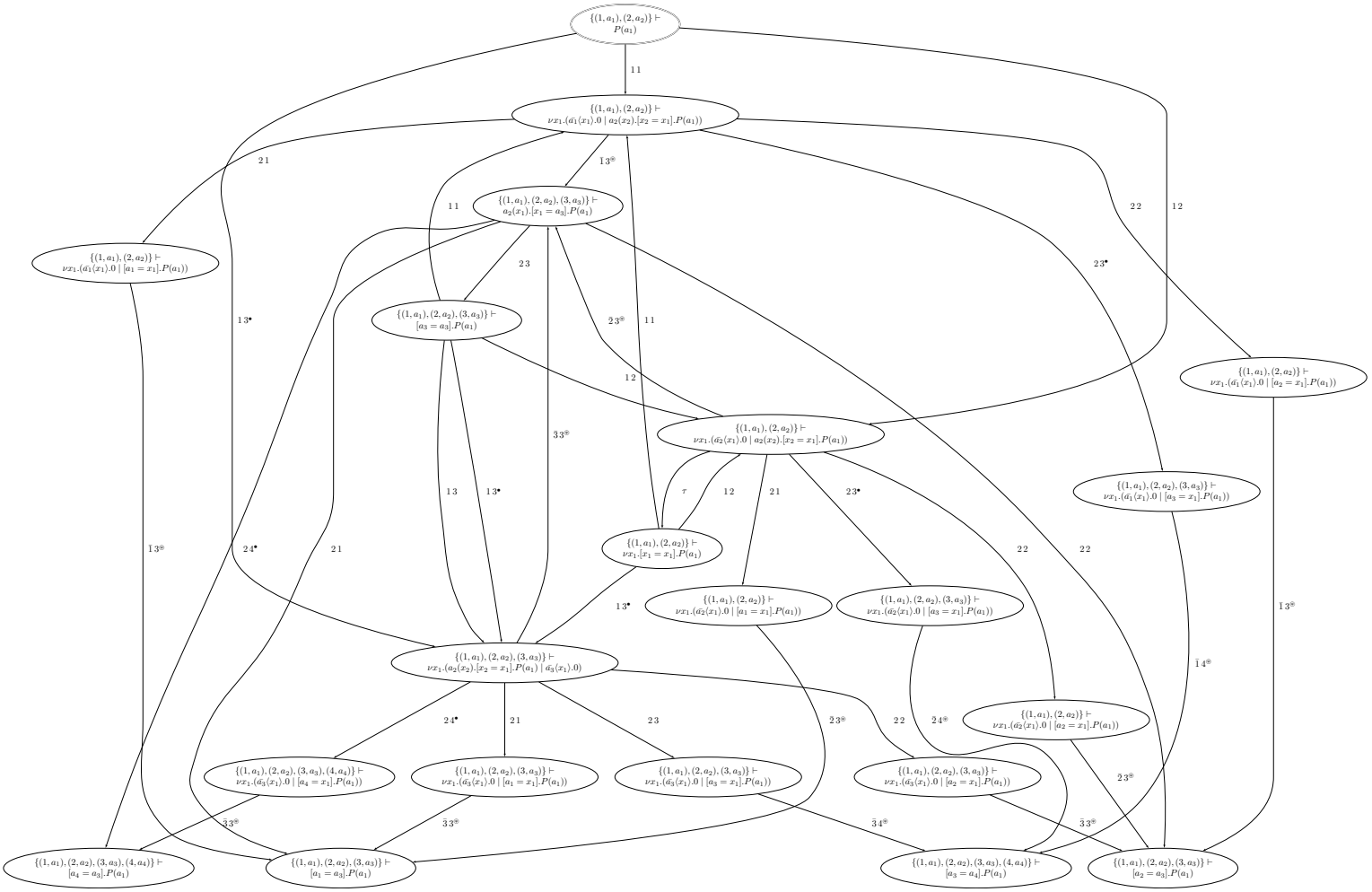


Fig. 8.4: *vk-fin-st3* visualisation.

vk-fin-st4

```
P(a) =  a(x).$y.(x'<y>.0  |  P(y))
P(a)
```

Listing 8.13: *vk-fin-st4* model.

```
s0 = {(1,#1)} |- P(#1)
s0  1 1   s1 = {(1,#1)} |- $&1.(#1'<&1>.0 | P(&1))
s0  1 1*  s1 = {(1,#1)} |- $&1.(#1'<&1>.0 | P(&1))
s1  1'1^  s0 = {(1,#1)} |- P(#1)
```

Listing 8.14: *vk-fin-st4* LTS.

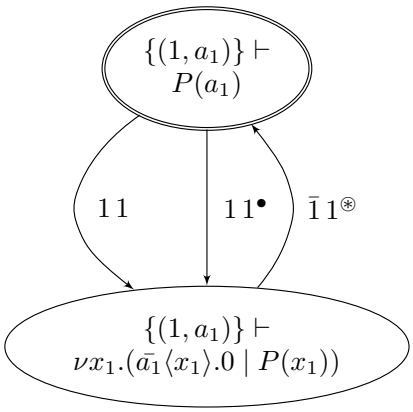


Fig. 8.5: *vk-fin-st4* visualisation.

```
states explored      2
states generated     3
states unique        2
transitions          3
time I/O             301.4µs
time LTS generation  712.5µs
```

Listing 8.15: *vk-fin-st4* statistics.

Password system

This model is the secure version of the password system from example 7.3.

```
GenPass(requestNewPass) = requestNewPass(x). $pass. x'<pass>.0

KeepSecret(requestNewPass) = $p. requestNewPass'<p>. p(pass). (
    StoreSecret(pass) | TestSecret(pass) )

StoreSecret(pass) = $secret. pass'<secret>. StoreSecret(pass)

TestSecret(pass) = pub(x). pass(secret). (
    TestSecret(pass) + [x=secret] _BAD'<_BAD>.0 )

$requestNewPass. (
    GenPass(requestNewPass)  |  KeepSecret(requestNewPass) )
```

Listing 8.16: *password* model.

```
s0 = {(1,_BAD),(2,#1)} |- $&1.(GenPass(&1) | KeepSecret(&1))
s0  t     s1 = {(1,_BAD),(2,#1)} |- $&1.($&2.&1'<&2>.0 | &1(&3).(StoreSecret(&3) | TestSecret(&3)))
s1  t     s2 = {(1,_BAD),(2,#1)} |- $&1.(StoreSecret(&1) | TestSecret(&1))
s2  2 1   s3 = {(1,_BAD),(2,#1)} |- $&1.(&1(&2).(TestSecret(&1) + [_BAD=&2]_BAD'<_BAD>.0) | StoreSecret
      (&1))
s2  2 2   s4 = {(1,_BAD),(2,#1)} |- $&1.(&1(&2).(TestSecret(&1) + [#1=&2]_BAD'<_BAD>.0) | StoreSecret(&1)
      )
s2  2 3*  s5 = {(1,_BAD),(2,#1),(3,#2)} |- $&1.(&1(&2).(TestSecret(&1) + [#2=&2]_BAD'<_BAD>.0) |
      StoreSecret(&1))
s3  t     s6 = {(1,_BAD),(2,#1)} |- $&1.(($&2.[_BAD=&2]_BAD'<_BAD>.0 + TestSecret(&1)) | StoreSecret(&1))
s4  t     s7 = {(1,_BAD),(2,#1)} |- $&1.(($&2.[#1=&2]_BAD'<_BAD>.0 + TestSecret(&1)) | StoreSecret(&1))
s5  t     s8 = {(1,_BAD),(2,#1),(3,#2)} |- $&1.(($&2.[#2=&2]_BAD'<_BAD>.0 + TestSecret(&1)) | StoreSecret
      (&1))
s6  2 1   s3 = {(1,_BAD),(2,#1)} |- $&1.(&1(&2).(TestSecret(&1) + [_BAD=&2]_BAD'<_BAD>.0) | StoreSecret
      (&1))
s6  2 2   s4 = {(1,_BAD),(2,#1)} |- $&1.(&1(&2).(TestSecret(&1) + [#1=&2]_BAD'<_BAD>.0) | StoreSecret(&1)
      )
s6  2 3*  s5 = {(1,_BAD),(2,#1),(3,#2)} |- $&1.(&1(&2).(TestSecret(&1) + [#2=&2]_BAD'<_BAD>.0) |
      StoreSecret(&1))
s7  2 1   s3 = {(1,_BAD),(2,#1)} |- $&1.(&1(&2).(TestSecret(&1) + [_BAD=&2]_BAD'<_BAD>.0) | StoreSecret
      (&1))
s7  2 2   s4 = {(1,_BAD),(2,#1)} |- $&1.(&1(&2).(TestSecret(&1) + [#1=&2]_BAD'<_BAD>.0) | StoreSecret(&1)
      )
s7  2 3*  s5 = {(1,_BAD),(2,#1),(3,#2)} |- $&1.(&1(&2).(TestSecret(&1) + [#2=&2]_BAD'<_BAD>.0) |
      StoreSecret(&1))
s8  2 1   s3 = {(1,_BAD),(2,#1)} |- $&1.(&1(&2).(TestSecret(&1) + [_BAD=&2]_BAD'<_BAD>.0) | StoreSecret
      (&1))
s8  2 2   s4 = {(1,_BAD),(2,#1)} |- $&1.(&1(&2).(TestSecret(&1) + [#1=&2]_BAD'<_BAD>.0) | StoreSecret(&1)
      )
s8  2 3   s5 = {(1,_BAD),(2,#1),(3,#2)} |- $&1.(&1(&2).(TestSecret(&1) + [#2=&2]_BAD'<_BAD>.0) |
      StoreSecret(&1))
s8  2 3*  s5 = {(1,_BAD),(2,#1),(3,#2)} |- $&1.(&1(&2).(TestSecret(&1) + [#2=&2]_BAD'<_BAD>.0) |
      StoreSecret(&1))
```

Listing 8.17: *password* LTS.

```
states explored      9
states generated     18
states unique        9
transitions          18
time I/O             353.9µs
time LTS generation  4.7428ms
```

Listing 8.18: *password* statistics.

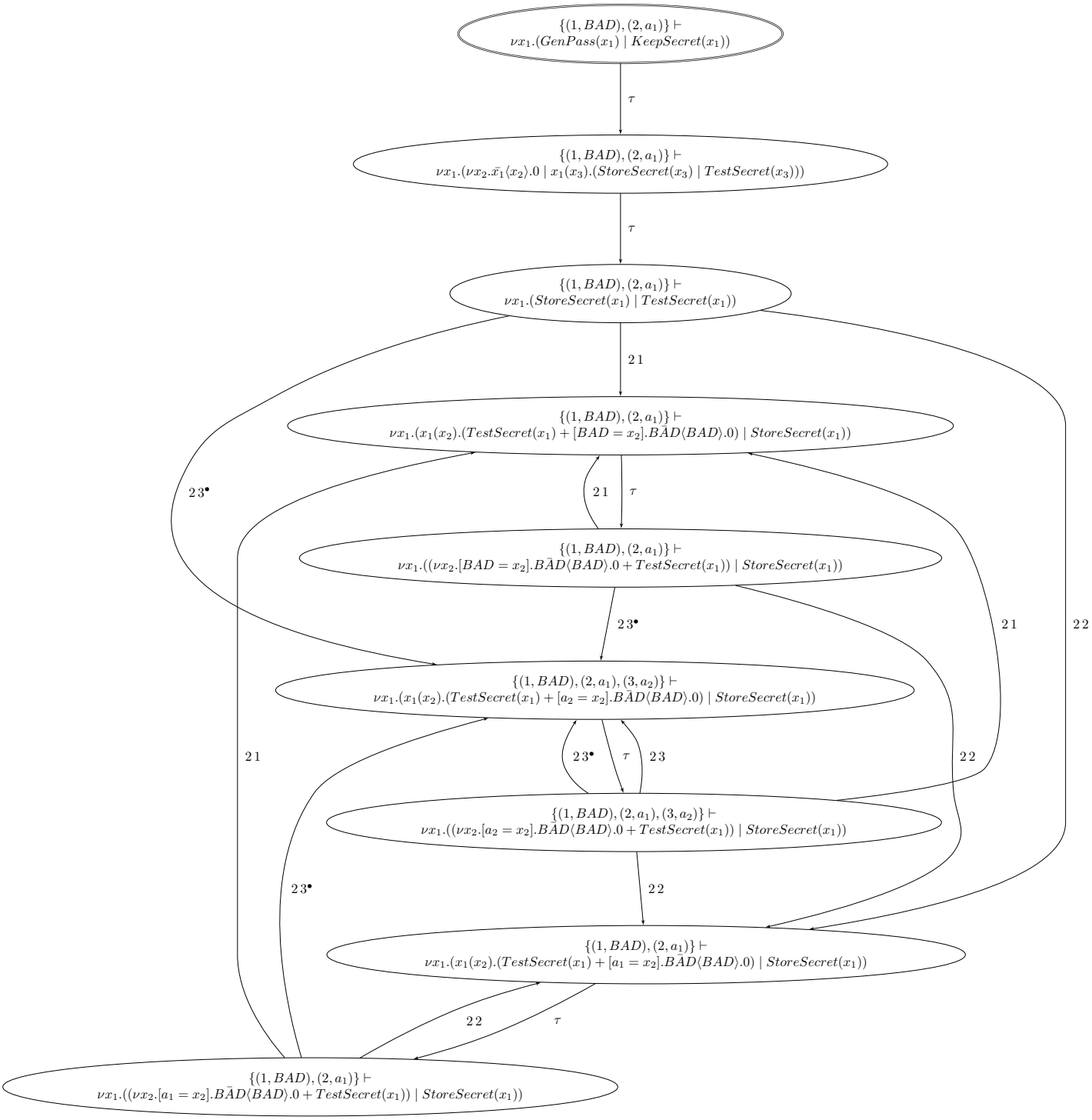


Fig. 8.6: *password* visualisation.

8.2 Infinite-state models

For infinite-state models, it is not possible to represent the LTS finitely. However, we can investigate the performance of the LTS generation by assessing the time taken at different state exploration limits.

In measuring the performance of the program, the CPU used is an *Intel Core* i7-9750H with 16GB of RAM running *Ubuntu* 18.04. For the run times shown, the program is run for a total of five times and the average time is calculated.

For the purpose of graphing the performance, in an LTS we define N to be the set of states $\mathcal{Q}$ and transitions $\mathcal{T}$, i.e., $N = \mathcal{Q} \cup \mathcal{T}$. In a graph $G = (V, E)$, this is the set of vertices and edges $(V+E)$.

vk-inf-reg2

'inf-reg' means that the number of registers grow indefinitely and as result there are an infinite number of states.

```
P = $x. a'<x>.( $y. y'<x>.0 | P )
P
```

Listing 8.19: *vk-inf-reg2* model.

```
s3  1'5^  s4 = {(1,#1),(2,#2),(3,#3),(4,#4),(5,#5)} |- ($&1.&1'<#2>.0 | ($&2.&2'<#3>.0 | ($
    &3.&3'<#4>.0 | ($&4.&4'<#5>.0 | P))))
```

Listing 8.20: *vk-inf-reg2* sample state.

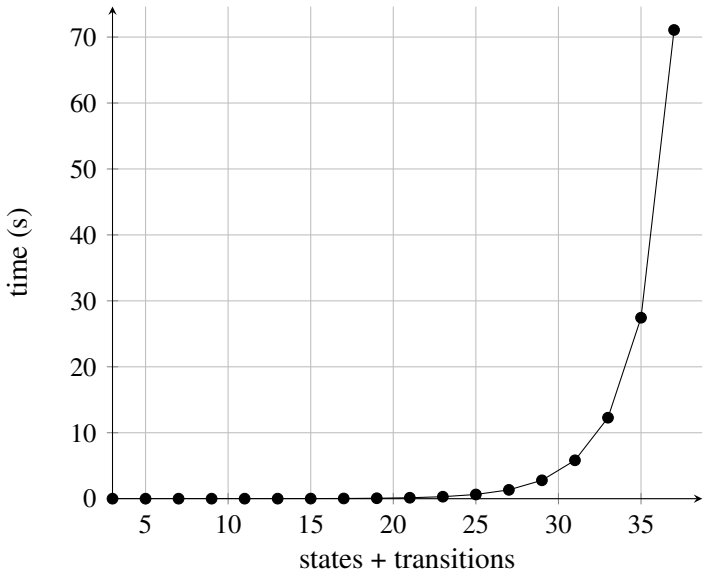


Fig. 8.7: *vk-inf-reg2* performance.

The time taken to generate the LTS grows exponentially with the number of states and transitions gathered. The 'elbow' of the graph lies somewhere near 33 N. From this point, the time taken to generate the next states increases significantly. This is mainly due to nested parallel elements. For parallel elements $P \mid Q$, we call `trans()` four times using the following states: $\sigma \vdash P$, $\sigma \vdash Q$, $(\sharp + \sigma) \vdash P$, and $(\sharp + \sigma) \vdash Q$ from the transition rules Par, Comm and Close, and their symmetric counterparts. `trans()` is called recursively a number of times, resulting in the rapid time growth rate.

The state at the elbow of the graph contains 16 processes in parallel. Therefore, with states with more than 16 processes and beyond in parallel, it is expected that the time taken to generate the next states will be computationally expensive.

The following table summarises the cost of recursive calls to `trans()` corresponding to each element.

element		*no. of calls to* `trans()`	*configurations passed to* `trans()`
input	$a(b).P$	0	
output	$\bar{a}\langle b\rangle.P$	0	
equality	$[a{=}a]P$	1	$\sigma \vdash P$
inequality	$[a{\neq}b]P$	1	$\sigma \vdash P$
restriction	$\nu a.P$	1	$(\sigma + a) \vdash P$
summation	$P + Q$	2	$\sigma \vdash P,\ \sigma \vdash Q$
composition	$P \mid Q$	4	$\sigma \vdash P,\ \sigma \vdash Q,\ (\sharp + \sigma) \vdash P,\ (\sharp + \sigma) \vdash Q$
process	$p(\vec{a})$	1	$\sigma \vdash P\{\vec{a}/\vec{b}\}$
inaction	0	0	

Tab. 8.1: Elements and their number of calls to `trans()` and their configurations passed to `trans()`.

We can see that the two binary elements composition and summation are the most costly elements at 4 and 2 calls to `trans()`, respectively.

vk-inf-reg1

```
P = $x. a'<x>.( x(y).0  |  P )
P
```

Listing 8.21: *vk-inf-reg1* model.

```
s19  1'6^  s15 = {(1,#1),(2,#2),(3,#3),(4,#4),(5,#5),(6,#6)} |- (#2(&1).0 | (#3(&2).0 |
     (#4(&3).0 | (#5(&4).0 | (#6(&5).0 | P)))))
```

Listing 8.22: *vk-inf-reg1* sample state.

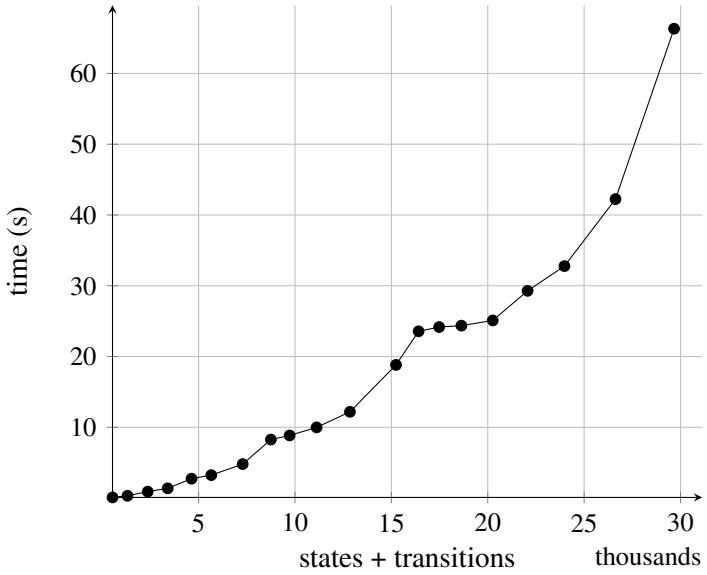


Fig. 8.8: *vk-inf-reg1* performance.

The shape of the graph is exponential but its pattern is staggered. Since the experiment is repeated five times, we offset any potential observational errors. Therefore, the pattern found is consistent.

Note the small increments in time taken between 535 and 7289 N, and then a sudden jump in time taken at 8750 N. The same occurs from 16414 to 20261 N, and then a jump at 22067 N. This pattern occurs because during the linear step increases (535-7289 N, 16414-20261 N), many states and transitions are found from the states in the queue, so the pattern is linear. However, once all of these states are dequeued, only a small number of states and transitions are found from those in the queue. The small size of N does not reflect the increase in time taken to generate them, hence the sharp rise in the graph.

vk-inf-st3

```
P = a(x). ( P | P )
P
```

Listing 8.23: *vk-inf-st3* model.

```
s12  1 2*  s13 = {(1,#1)} |- (P | (P | (P | (P | (P | (P | (P | (P | (P | (P | (P | (P |
    P))))))))))))
```

Listing 8.24: *vk-inf-st3* sample state.

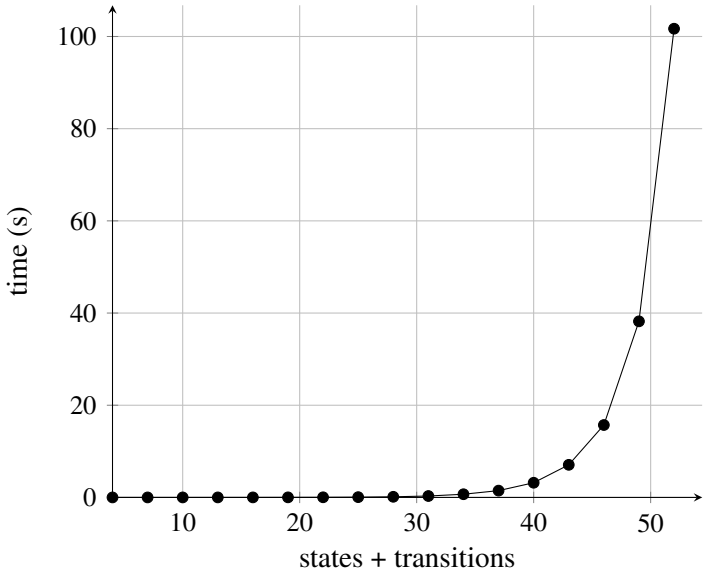


Fig. 8.9: *vk-inf-st3* performance.

The growth pattern is exponential due to the increasing number of parallel elements. The elbow of the graph at 46 N contains 16 processes in parallel, which is consistent with the number of parallels as seen before at this point. Therefore, gathering next states with more than 16 processes in parallel becomes highly expensive.

vk-inf-st4

```
P(a) = a(x). ( P(x) | P(x) )
P(a)
```

Listing 8.25: *vk-inf-st4* model.

```
s196  6 7*  s553 = {(1,#1),(2,#2),(3,#3),(4,#4),(5,#5),(6,#6),(7,#7)} |- (P(#1) | (P(#2) | (P
    (#3) | (P(#4) | (P(#5) | (P(#6) | (P(#7) | P(#7)))))))
```

Listing 8.26: *vk-inf-st4* sample state.

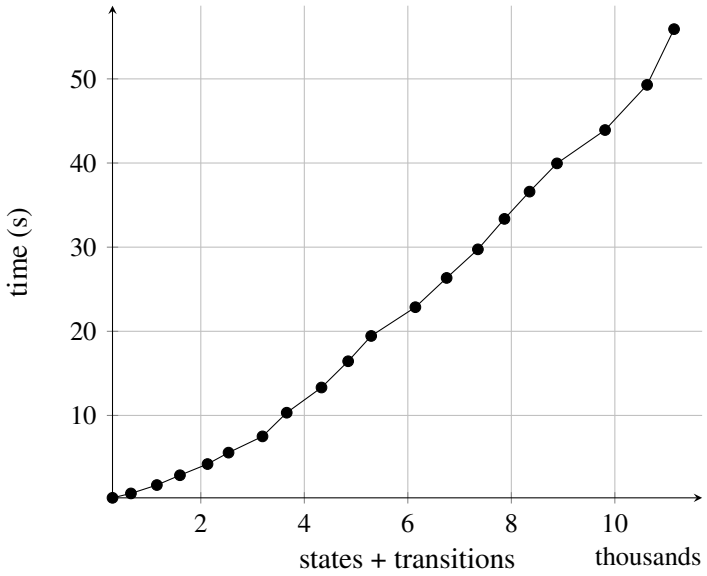


Fig. 8.10: *vk-inf-st4* performance.

The time taken to generate the LTS grows linearly with the number of states and transitions gathered. The expression is quite similar to the previous *vk-inf-st3* model, except that this process definition takes in a name. It seems perplexing that the two expressions result in different growth patterns.

The reason for this is due to the different frontier sizes. The frontier of an LTS is a state where the next states are generated. For *vk-inf-st3*, the frontier size grows by 2 for each state explored. For *vk-inf-st4*, the frontier size grows exponentially for each state explored. Therefore, these new states and transitions found add to the size of N and maintain the graph at a linear pattern. To demonstrate, s_{10} for *vk-inf-st3* is

```
s9  1 1   s10 = {(1,#1)} |- (P | (P | (P | (P | (P | (P | (P | (P | (P | P)
    )))))))))
```

and for *vk-inf-st4*, the state is

```
s4  1 1   s10 = {(1,#1)} |- (P(#1) | (P(#1) | (P(#1) | (P(#1) | P(#1)))))
```

Note how for *vk-inf-st4*, s_{10} is generated from s_4, indicating that new states are added to the queue at an early stage. For *vk-inf-st3*, s_{10} is generated from s_9, indicating a small frontier size. Essentially, each state generation for *vk-inf-st3* yields only 2 new

states (regardless of the number of parallel elements), whereas for *vk-inf-st4*, each state exploration will yield an increasing number of states. So even with the additional time taken due to the number of parallels, its number of new states and transitions found will offset the generation time. The next models will make this clearer.

vk-inf-st1

```
P = a'<a>.( P  |  $y. (y'<y>.0  |  y(x).0) )
P
```

Listing 8.27: *vk-inf-st1* model.

```
s3  1'1   s4 = {(1,#1)} |- ($&1.(&1'<&1>.0 | &1(&2).0) | ($&3.(&3'<&3>.0 | &3(&4).0) | ($
    &5.(&5'<&5>.0 | &5(&6).0) | ($&7.(&7'<&7>.0 | &7(&8).0) | P))))
```

Listing 8.28: *vk-inf-st1* sample state.

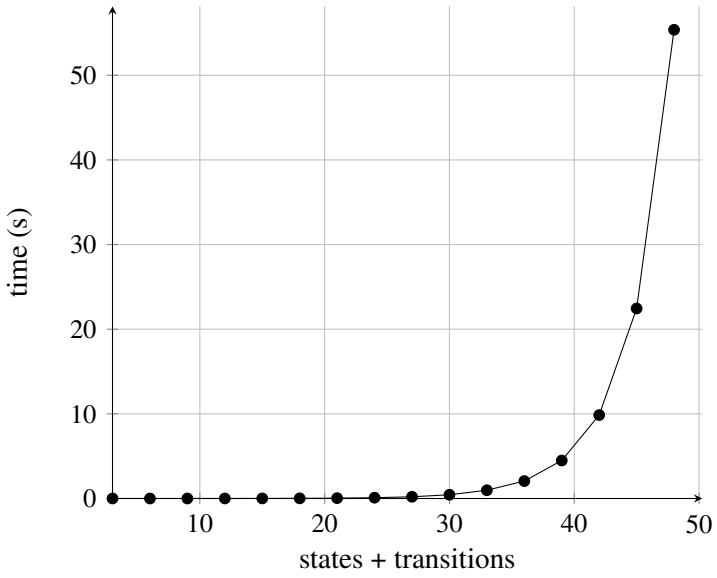


Fig. 8.11: *vk-inf-st1* performance.

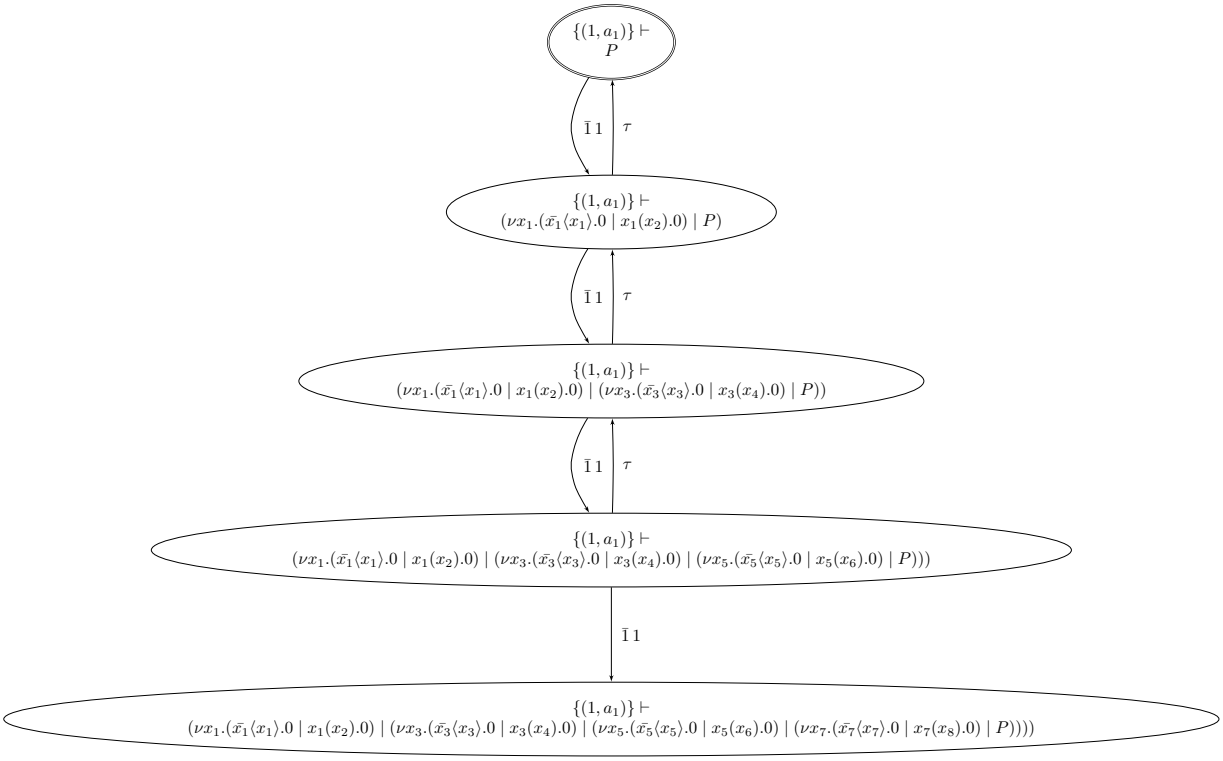


Fig. 8.12: *vk-inf-st1* visualisation with 4 states explored.

With the visualisation of 4 states explored, the frontier of the LTS can be clearly seen. Only one next state is found for each state explored, even though the number of parallel processes increases at each state. The calling of `trans()` becomes more computationally expensive at each exploration, contributing to the exponential trend of the graph.

vk-inf-st2

```
P = a(x).( P  |  $y. x'<y>.0 )
P
```

Listing 8.29: *vk-inf-st2* model.

```
s99  1 5*  s386 = {(1,#1),(2,#2),(3,#3),(4,#4),(5,#5)} |- ($&1.#2'<&1>.0 | ($&2.#3'<&2>.0 | ($
     &3.#4'<&3>.0 | ($&4.#2'<&4>.0 | ($&5.#5'<&5>.0 | P)))))
```

Listing 8.30: *vk-inf-st2* sample state.

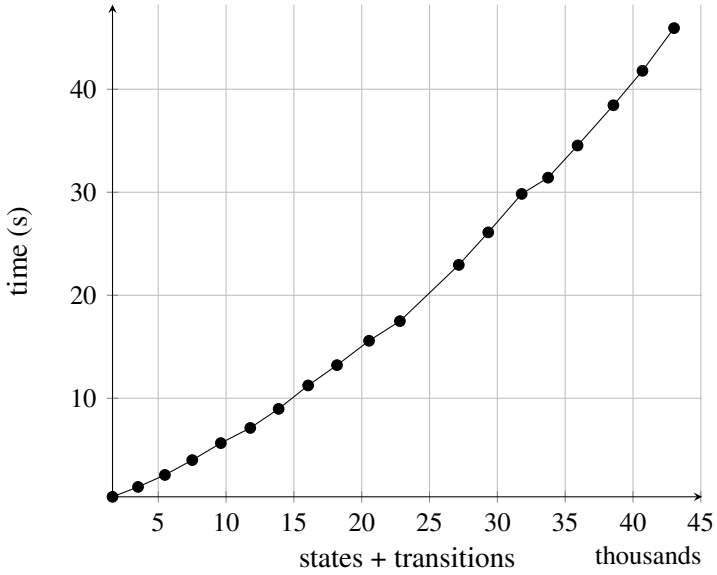


Fig. 8.13: *vk-inf-st2* performance.

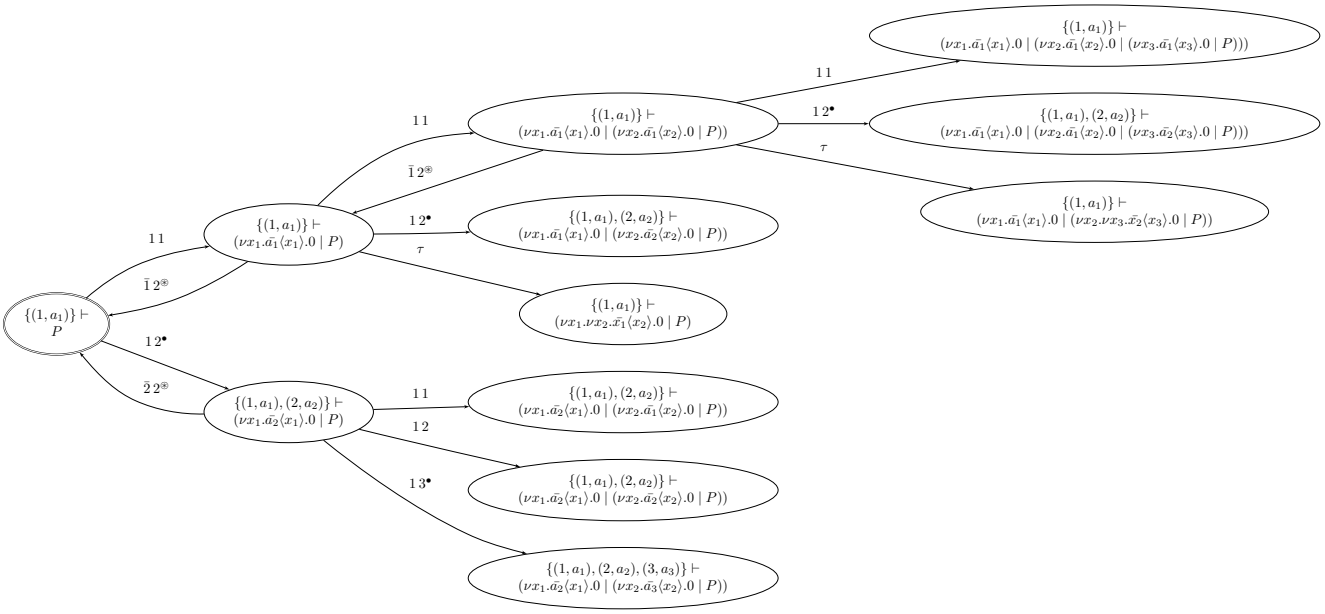


Fig. 8.14: *vk-inf-st2* visualisation with 4 states explored.

From the visualisation, the frontier increases exponentially in size for this model. Exploring a state finds many more new states (and transitions), which therefore increases the size of N. This makes it such that for every new state explored, the number of states and transitions found from calling `trans()` scales with the number of parallel processes. As a result, the shape of the graph becomes linear.

Conclusion

9

The goal of the dissertation was to generate an LTS from pi-calculus models through the use of fresh-register automata. This was fully achieved through the construction of a tool which implemented the underlying theory to perform this.

9.1 Achievements

To our knowledge, this is the only implementation of an LTS generation tool of pi-calculus models by using fresh-register automata. In 2011, Tzevelekos defined FRA's and the FRA transition relation for the pi-calculus. The dissertation aimed to implement Tzevelekos' theory and this was realised.

We began by specifying the $\times\pi$-calculus grammar, parsing its language, and representing the models internally. From these input models, we translated them to FRA's through a transition relation. The original $\times\pi$-calculus transition relation was revised to improve practicality and each rule was thoroughly implemented. Using structural congruence as a starting point, we expanded the definition of congruence in order to accommodate the definition of configurations in FRA's. We devised algorithms to normalise configurations, thereby enabling the detection of equivalent states. By exploring the states breadth-first using the transition relation and normalising each state, we constructed a minimised directed graph, thus forming the LTS of a model. We demonstrated the viability of using the generated LTS as a verification tool by proving or disproving whether a target state can be reached. Finally, we showcased LTS's from a number of finite and infinite-state models generated using the tool.

The tool produced is encapsulated in an intuitive command-line program, which allows for maximum compatibility and scripting capabilities. Hopefully, this tool will aid researchers and learners in this area and is used as a stepping stone for building model checking tools for wider applications.

9.2 Challenges

During the implementation of the transition relation, it was not apparent that certain transition rules could be applied at different points of the derivation tree due to intermediate stages. This led to confusion in terms of the design and approach. There was discussion about having a tentative LTS and applying the DBL rules on the second pass. This approach would have been more complex, as well as being more algorithmically inefficient. In the end, the rules were reviewed and modified to simplify the implementation.

Structural congruence with processes was well established by Milner. However, equivalence needed to be extended to configurations, which contain both the registers and the process. This was not mentioned in the FRA paper. Therefore, we developed and formalised the definition of configuration congruence.

The idea of garbage collection was not referred to in the FRA paper and its feasibility required further investigation. Koutavas theorised that removing names that did not appear in the process from the registers did not interfere with the LTS generation. However, transition rules like the revised INP1 rule depended on the contents of the registers. After an exchange of emails between Koutavas and Tzevelekos, Tzevelekos confirmed the viability of garbage collection, which would result in a bisimilar LTS under the $\times\pi$-bisimilarity notion. As a result, GC could be implemented without compromising correctness.

Models containing states with many parallel processes (>16) are subject to performance bottlenecks due to nested calls to `trans()`. The revised transition relation was implemented faithfully with no little to no deviations. Perhaps the transition rules could have been rewritten with optimisations which reduces the number of calls to `trans()`. However, this was not performed because there was a trade-off between correctness and performance, and we chose the former. Heavy modification to the rules could have resulted in an incorrect LTS.

Throughout the development of the LTS generation tool, a number of issues and bugs were encountered, which were eventually resolved. The use of unit tests and writing clean code helped to uncover instances of bugs. Numerous parts of the codebase were continuously iterated upon in terms of its structure, algorithms, and data structures.

9.3 Future work

At present, state reachability checks are performed by manually encoding the target state and searching for the corresponding action transition. This verification procedure can be formalised through the use of a properties language. An example is a modal logic like LTL [29] which encodes conditions of the LTS, e.g., a state will eventually be reached. Future work would examine how a properties language like LTL could be integrated with the tool.

It was anticipated that this dissertation would not cover the scope of the broader research project. However, the foundational step of generating a labelled transition system from pi-calculus models was achieved. The next stage of the research project is bisimulation. From two LTS's generated using the tool, can a bisimulation be found between these two systems? Since the LTS modelled by the π-calculus is represented by FRA's, we need to use Tzevelekos' definition of $\times\pi$-bisimulation. Known algorithms for solving standard bisimulations like weak and strong exist, including the Paige-Tarjan algorithm [30] and Kanellakis-Smolka algorithms [31]. However, these algorithms would need to be modified in order to work for $\times\pi$-bisimulations. Future work would focus on investigating how these algorithms would operate with $\times\pi$-bisimulations.

Source Code

The source code, tests, and documentation for this tool can be found at `https://github.com/sengleung/pifra`.

Bibliography

[1] Robin Milner. *Communicating and Mobile Systems: The Pi-Calculus*. USA: Cambridge University Press, 1999. ISBN: 0521658691 (cit. on pp. 1, 4, 7, 9, 10, 12–16, 29, 48).

[2] Sergey Berezin, Sérgio Vale Aguiar Campos, and Edmund M. Clarke. "Compositional Reasoning in Model Checking". In: *Revised Lectures from the International Symposium on Compositionality: The Significant Difference*. COMPOS97. Berlin, Heidelberg: Springer-Verlag, 1997, pp. 81–102. ISBN: 3540654933 (cit. on p. 1).

[3] Gerard J. Holzmann. "The model checker SPIN". In: *IEEE Transactions on software engineering* 23.5 (1997), pp. 279–295 (cit. on pp. 2, 6).

[4] L. Lamport. "Proving the Correctness of Multiprocess Programs". In: *IEEE Trans. Softw. Eng.* 3.2 (Mar. 1977), pp. 125–143. ISSN: 0098-5589. DOI: 10.1109/TSE.1977.229904. URL: https://doi.org/10.1109/TSE.1977.229904 (cit. on p. 3).

[5] Nikos Tzevelekos. "Fresh-Register Automata". In: *SIGPLAN Not.* 46.1 (Jan. 2011), pp. 295–306. ISSN: 0362-1340. DOI: 10.1145/1925844.1926420. URL: https://doi.org/10.1145/1925844.1926420 (cit. on pp. 4, 16–20, 31, 32, 50, 70).

[6] Edmund M. Clarke, William Klieber, Milo Nováek, and Paolo Zuliani. "Model Checking and the State Explosion Problem". In: *Tools for Practical Software Verification: LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*. Ed. by Bertrand Meyer and Martin Nordio. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–30. ISBN: 978-3-642-35746-6. DOI: 10.1007/978-3-642-35746-6_1. URL: https://doi.org/10.1007/978-3-642-35746-6_1 (cit. on p. 6).

[7] Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, et al. *The Coq Proof Assistant, Reference Manual, Version 5.10*. Research Report RT-0177. Projet COQ. INRIA, 1995, p. 185. URL: https://hal.inria.fr/inria-00069994 (cit. on p. 6).

[8] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Berlin, Heidelberg: Springer-Verlag, 2002. ISBN: 3540433767 (cit. on p. 6).

[9] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: https://doi.org/10.1145/363235.363259 (cit. on p. 6).

[10] Eugene W. Stark. "A Proof Technique for Rely/Guarantee Properties". In: *Proceedings of the Fifth Conference on Foundations of Software Technology and Theoretical Computer Science*. Berlin, Heidelberg: Springer-Verlag, 1985, pp. 369–391. ISBN: 3540160426 (cit. on p. 6).

[11] C. A. R. Hoare. *Communicating Sequential Processes*. USA: Prentice-Hall, Inc., 1985. ISBN: 0131532715 (cit. on p. 7).

[13] Robin Milner. *A Calculus of Communicating Systems*. Berlin, Heidelberg: Springer-Verlag, 1982. ISBN: 0387102353 (cit. on p. 7).

[14] Matthew Hennessy and Robin Milner. "On Observing Nondeterminism and Concurrency". In: *Proceedings of the 7th Colloquium on Automata, Languages and Programming*. Berlin, Heidelberg: Springer-Verlag, 1980, pp. 299–309. ISBN: 3540100032 (cit. on p. 7).

[15] Sören Holmström. "A Refinement Calculus for Specifications in Hennessy-Milner Logic with Recursion". In: *Form. Asp. Comput.* 1.1 (Mar. 1989), pp. 242–272. ISSN: 0934-5043. DOI: 10.1007/BF01887208. URL: https://doi.org/10.1007/BF01887208 (cit. on p. 7).

[16] Martn Abadi and Andrew D. Gordon. "A Calculus for Cryptographic Protocols: The Spi Calculus". In: *Proceedings of the 4th ACM Conference on Computer and Communications Security*. CCS 97. Zurich, Switzerland: Association for Computing Machinery, 1997, pp. 36–47. ISBN: 0897919122. DOI: 10.1145/266420.266432. URL: https://doi.org/10.1145/266420.266432 (cit. on p. 11).

[17] Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. USA: Cambridge University Press, 2011. ISBN: 1107003636 (cit. on p. 16).

[18] Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii. "Environmental Bisimulations for Higher-Order Languages". In: *ACM Trans. Program. Lang. Syst.* 33.1 (Jan. 2011). ISSN: 0164-0925. DOI: 10.1145/1889997.1890002. URL: https://doi.org/10.1145/1889997.1890002 (cit. on p. 16).

[19] Davide Sangiorgi and Robin Milner. "The Problem of "Weak Bisimulation up to". In: *Proceedings of the Third International Conference on Concurrency Theory*. CONCUR 92. Berlin, Heidelberg: Springer-Verlag, 1992, pp. 32–46. ISBN: 3540558225 (cit. on p. 16).

[20] R. M. Needham. "Names". In: *Distributed Systems*. New York, NY, USA: Association for Computing Machinery, 1990, pp. 89–101. ISBN: 0201416603. URL: https://doi.org/10.1145/90417.90741 (cit. on p. 17).

[21] Michael Kaminski and Nissim Francez. "Finite-Memory Automata". In: *Theor. Comput. Sci.* 134.2 (Nov. 1994), pp. 329–363. ISSN: 0304-3975. DOI: 10.1016/0304-3975(94)90242-9. URL: https://doi.org/10.1016/0304-3975(94)90242-9 (cit. on p. 17).

[24] N.G de Bruijn. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". In: *Indagationes Mathematicae (Proceedings)* 75.5 (1972), pp. 381–392. ISSN: 1385-7258. DOI: https://doi.org/10.1016/1385-7258(72)90034-0 (cit. on p. 29).

[25] Edsko de Vries and Vasileios Koutavas. "Locally Nameless Permutation Types". In: *CoRR* abs/1710.08444 (2017). arXiv: 1710.08444. URL: http://arxiv.org/abs/1710.08444 (cit. on p. 49).

[28] Fred B. Schneider. "Enforceable Security Policies". In: *ACM Trans. Inf. Syst. Secur.* 3.1 (Feb. 2000), pp. 30–50. ISSN: 1094-9224. DOI: 10.1145/353323.353382. URL: https://doi.org/10.1145/353323.353382 (cit. on p. 69).

[29] Amir Pnueli. "The Temporal Logic of Programs". In: *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*. SFCS 77. USA: IEEE Computer Society, 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32. URL: https://doi.org/10.1109/SFCS.1977.32 (cit. on p. 89).

[30] Robert Paige and Robert E. Tarjan. “Three Partition Refinement Algorithms”. In: *SIAM J. Comput.* 16.6 (Dec. 1987), pp. 973–989. ISSN: 0097-5397. DOI: 10.1137/0216062. URL: https://doi.org/10.1137/0216062 (cit. on p. 89).

[31] Paris C. Kanellakis and Scott A. Smolka. “CCS Expressions, Finite State Processes, and Three Problems of Equivalence”. In: *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*. PODC 83. Montreal, Quebec, Canada: Association for Computing Machinery, 1983, pp. 228–240. ISBN: 0897911105. DOI: 10.1145/800221.806724. URL: https://doi.org/10.1145/800221.806724 (cit. on p. 89).

Webpages

[12] Go authors. *Language Design FAQ: Why build concurrency on the ideas of CSP?* URL: https://golang.org/doc/faq#csp (visited on Mar. 20, 2020) (cit. on p. 7).

[22] Colm Networks. *Ragel*. URL: http://www.colm.net/open-source/ragel (visited on Feb. 23, 2020) (cit. on p. 21).

[23] goyacc authors. *goyacc*. URL: https://pkg.go.dev/golang.org/x/tools/cmd/goyacc (visited on Feb. 23, 2020) (cit. on p. 21).

[26] AT&T Labs Research and contributors. *GraphViz DOT Language*. URL: https://www.graphviz.org/doc/info/lang.html (visited on Mar. 5, 2020) (cit. on p. 61).

[27] Kjell Magne Fauske. *dot2tex*. URL: https://dot2tex.readthedocs.io/ (visited on Apr. 5, 2020) (cit. on p. 63).

List of Figures

List of Tables

Listings