Bachelor Project Thesis

Computing Science

An Exercise on Program Equivalence for Message-passing Processes

Bianca Udrescu
S3136728

Supervised by
Jorge A. Pérez
Gerard Renardel de Lavalette

October 1, 2019
University of Groningen
An exercise on program equivalence for message-passing processes

Bianca Udrescu
University of Groningen, The Netherlands
S3136728

Contents

1 Introduction ........................................ 1
    1.1 Structure ...................................... 1

2 Preliminaries ...................................... 2
    2.1 Higher-Order π-Calculus ...................... 2
        2.1.1 Syntax and Semantics ................. 2
        2.1.2 Labelled Transition System for Processes .......... 3
    2.2 Program Equivalence .......................... 4
        2.2.1 Weak Simulation ........................ 4
    2.3 Program Decomposition ...................... 5
        2.3.1 Session Types for HO .................. 5
        2.3.2 Preliminary Definitions and Main Idea .......... 6

3 An Exercise on Program Equivalence .......... 8

4 Conclusion ..................................... 18
Abstract

This paper represents an extension to the research output extracted from the paper “Minimal Session Types” [1] authored by A. Arslanagić, J. A. Pérez, and E. Voogd in the context of the bachelors project presented in the final year of the Computing Science degree.

Delving into one of the fundamental aspects or concurrency, this project analyzes equivalence between message-passing programs by using a popular technique in process calculi called simulation.

Following the recent research in the previously mentioned paper, it was concluded that certain message-passing programs can be decomposed into smaller pieces denoted as trios or concerts. A program $P$, will be decomposed into another program $D(P)$ typed using minimal session types by applying the breakdown function $B^k_x(\cdot)$ on its inner component values.

My scientific contribution to the fundamental computing group is a proof showcasing the equivalence relation between $P$ and its decomposition, $D(P)$.

As a prerequisite to the deliverable, my research includes an in-depth look into mathematical models of processes including $\pi$-calculus, higher-order $\pi$-calculus and the analysis of program equivalence techniques including weak and strong simulation and bi-simulation.

1 Introduction

Concurrency is becoming one of the most important topics in the field of IT ever since the use of an increased number of cores per computer and the demand for highly distributed systems.

However, making the best use out of this technology advancement has proved to be a non-trivial goal due to the non-deterministic behaviour of processes and their interactions. The already stated challenge brings us to the main research question regarding this paper: When do two implementations of the same program behave the same in a context involving concurrency?

In order to express processes and concurrency we have used a process calculus called $\pi$-calculus [5]. Process calculi represent small programming-like languages that have been created to model the actions, communications, and synchronizations between processes. Those are a key element in concurrency as it is mandatory to be able to represent the continuous structural reconfiguration that happens in communicating systems.

Several mathematical models of proving program equivalence have been studied, nonetheless the one we have settled for is weak simulation. Intuitively, we aim to develop every possible state a process could evolve into and verify if another implementation of the same process can mimic those actions. Formally, the precedent can be proved by constructing a binary relation $R$ between the two.

Following the recent research done by the fundamental computing group at the University of Groningen, it was concluded that certain message-passing programs can be decomposed into smaller pieces denoted as trios or concerts. My scientific contribution to the fundamental computing group is proving the equivalence relation between a process and its decomposition.

1.1 Structure

The paper has been structured in the following manner:

The preliminaries containing the mandatory research necessary to complete the scope of this project is detailed in Section 2 and includes the language used to express the two processes,
the definition of weak simulation along with a description of minimal session types and of the breakdown function.
Section 3 details every step taken to complete the similarity proof alongside visual representations and tables containing the transition states developed for the processes. Finally, Section 4 specifies possible future work that could expand my research along with an overview of challenges and overall conclusions regarding the paper.

2 Preliminaries

2.1 Higher-Order \( \pi \)Calculus

Concurrent computation in systems that dynamically change the configuration of their communication has been a pressing subject in the field of computing science. Process calculi are small programming-like languages that have done a great venture in representing this continuous structural reconfiguration called mobility. In this particular case, higher-order \( \pi \)-calculus has been exploited due to its simplicity: it allows for agents to be passed as values during communication, which include variables and abstractions but it does not allow for recursion or passing names.

The present paper uses the syntax, semantics, and type system of the higher-order process calculus for session-based concurrency studied by Kouzapas et al [2].

2.1.1 Syntax and Semantics

This subsection defines the core of the syntax of names, variables, values, and HO processes that will be further utilized:

\[
\begin{align*}
 n, m & := a, b | s, s' & u, w & := n | x, y, z & V, W & := x, y, z | \lambda x . P \\
 P, Q & := u!(V).P | u?(x).P | V u | P | Q | (\nu n)P | 0
\end{align*}
\]

The letters \( a, b, c, \ldots \) stand for shared names which are used in unrestricted interactions, meaning that a channel can be used more than once by multiple processes not containing parallel composition.

The letters \( s, s', \ldots \) are to represent session names. In comparison to shared names, they are relevant in an linear and deterministic context that denotes a restricted one-time use of a channel.

Names and sessions are denoted by \( n, m \), variables by \( x, y, z, \ldots \) and values by \( V, W, \ldots \). Since we would like to exchange more than a singular value during the synchronization of sessions and in applications, we define a tuple of variables \( (x_1, \ldots, x_k) \) to be \( \vec{x} \) and the empty tuple to be \( \epsilon \). Same notations are used in the case of names and values.

Processes are denoted by \( P, Q, \ldots \). The output prefix \( u!(V).P \) is a channel end that can send a value \( V \) along \( u \). The input prefix \( u?(x).P \), is it’s dual and is ready to receive a value that will substitute the placeholder \( x \) and continue interaction as \( P \). In the instance in which the value is missing inside the brackets we assume that it is not relevant.

Process \( V u \) is the application which will substitute name \( u \) for \( x \) on abstraction \( V \) of the form \( \lambda x . P \) and will continue the interaction as \( P \).

Agent \( 0 \) is empty and represents that no further actions can be performed.

The composition \( P_1 | P_2 \) denotes two processes running in parallel and the name restriction \( (\nu n)P \) specifies that name \( n \) can only be accessed inside process \( P \). In the case when \( n \) stands for a session name both the name and it’s dual will be bound in \( P \).
We further define the structural congruence on processes, which stands for the smallest congruence on processes such that:

\[ P \parallel 0 \equiv P \quad P_1 \parallel P_2 \equiv P_1 \quad P_1 \parallel (P_2 \parallel P_3) \equiv (P_1 \parallel P_2) \parallel P_3 \quad (\nu n)0 \equiv 0 \]

\[ P \parallel (\nu n)Q \equiv (\nu n)(P \parallel Q) \quad (n \notin \text{fn}(P)) \quad P \equiv Q \text{ if } P \equiv_\alpha Q \]

\[ P \equiv_\alpha Q \text{ represents congruence between a process } P \text{ and a process } Q \text{ after the appliance of alpha-conversion on } Q. \]

All other notation is standard.

\[ \{ \text{Notation 1.} \}
\]

We use \( \text{fn}(P) \) to express the set of free names present in process \( P \) and \( \text{bn}(P) \) to express the set of bound names in \( P \).

\[ \{ \text{Remark 2.1 (Polyadic Communication).} \}
\]

The higher-order \( \pi \) calculus as described thus far allows for the exchgange of one value between processes. This is called monadic communication. However, we will be using higher-order \( \pi \)-calculus with polyadic communication, i.e., the exchange of multiple values over the same channel. The tuples of values are denoted by \( \tilde{V} \), with length \( k \geq 1 \). Polyadicity will appear in session synchronizations and applications, but not in synchronizations on shared names. The reduction rules for polyadic communication are presented as follows:

\[ (\lambda \tilde{x}. P) \tilde{u} \rightarrow P\{\tilde{x}/\tilde{u}\} \]

\[ s!\tilde{V}. P | \pi?\tilde{x}. Q \rightarrow P | Q\{\tilde{V}/\tilde{x}\} \]

\[ \{ \text{Notation 2.} \}
\]

\[ P\{\tilde{u}/\tilde{x}\} \text{ represents a substitution where names } \tilde{u} \text{ will replace names } \tilde{x} \text{ and continue interaction as } P. \]

This polyadic HO can be readily encoded into (monadic) HO [3]; therefore HO and polyadic HO will be used interchangeably in the present paper.

### 2.1.2 Labelled Transition System for Processes

In order to showcase similar behaviour between two processes, the transitions for each process have to be generated and represented as nodes in a tree.

We will use transitions of the form \( P \xrightarrow{\ell} P_i \) with the intuitive meaning that \( P \) is a process that preforms action \( \ell \) and evolves into \( P_i \).

Therefore, we define the interaction of processes with their environment using action labels \( \ell \):

\[ \ell ::= \tau \quad | \quad (\nu \tilde{m})n!(V) \quad | \quad n?(V) \]

Label \( \tau \) defines internal actions which are represented by synchronizations or by name application. Action \( (\nu \tilde{m})n!(V) \) denotes the sending of value \( V \) over channel \( n \) with a possible empty set of restricted names \( \tilde{m} \) (we may write \( n!(V) \) when \( \tilde{m} \) is empty). The action for value reception \( n?(V) \) is its dual. We write \( \text{fn}(\ell) \) and \( \text{bn}(\ell) \) to denote the sets of free/bound names in \( \ell \), respectively. The notation \( \ell_1 \doteq \ell_2 \) means that \( \ell_1 \) is the dual of \( \ell_2 \) and vice-versa.

Given \( \ell \neq \tau \), we say \( \ell \) is a visible action.

Figure 1 contains all the untyped LTS rules used in the present document extracted from [2]. A process containing an output prefix (\( n!(V) \)) can interact with the environment with an output action that sends value \( V \) (Rule \( \langle \text{Snd} \rangle \)). Dually, in Rule \( \langle \text{Rv} \rangle \) a process can receive an input of an arbitrary value \( V \). Rule \( \langle \text{Res} \rangle \) allows for a restricted process to perform an observable action as long as the restricted name does not occur free in the action. If
An exercise on program equivalence for message-passing processes

...a restricted name is present in the value of an output action, the process performs scope opening (Rule ⟨New⟩). Rule ⟨Tau⟩ states that two parallel processes which perform dual actions can synchronise by an internal transition. Rules ⟨ParL ⟩/⟨ParR ⟩ and ⟨Alpha⟩ present the behaviour of process actions under parallel composition and respectively α-conversion.

\[
\begin{align*}
\langle \text{App} \rangle & \quad (\lambda x. P) V \xrightarrow{\ell} P[V/x] \\
\langle \text{Snd} \rangle & \quad n!\langle V \rangle, P \xrightarrow{n!\ell} P \\
\langle \text{Rv} \rangle & \quad n?\langle x \rangle, P \xrightarrow{n?\ell} P[V/x] \\
\langle \text{Alpha} \rangle & \quad P \equiv_\alpha Q \quad Q \xrightarrow{\ell} P' \\
\langle \text{ParL} \rangle & \quad P \xrightarrow{\ell} P' \\
\langle \text{Res} \rangle & \quad P \xrightarrow{\ell} P' \\
\langle \text{New} \rangle & \quad P \xrightarrow{(\nu m)\text{fn}(V)} P' \\
\langle \text{ParR} \rangle & \quad n \notin \text{fn}(\ell) \\
\langle \text{Tau} \rangle & \quad (\nu n)P \xrightarrow{\ell} (\nu n)P'
\end{align*}
\]

\[\text{Fig. 1} \quad \text{The Untyped LTS for HO processes. We omit Rule } \langle \text{ParR} \rangle.\]

2.2 Program Equivalence

Studying one of the fundamental aspects or concurrency, this project analyzes equivalence between message-passing programs by using a popular technique in process calculi called simulation. There are different ways to represent how programs can mimic their interactions and in Section 2.2 we will present formal and intuitive definitions of the research regarding weak simulation: an asymmetric binary relation that allows to ignore silent transitions.

2.2.1 Weak Simulation

In process algebra, simulation is a highly researched and used mathematical technique to prove that processes exhibit equivalent behaviour.

Following the book by Sangiorgi [6], the intuition behind simulation was extracted. Denoting a process \( P \) and a process \( Q \), in order for the two to be similar the following must be true: If \( P \) performs an action \( \alpha \), \( Q \) must be able to perform the same action \( \alpha \) and the requirement is to hold for whatever processes \( Q \) and \( P \) evolve into. Since this is an asymmetric relation, the same reasoning does not necessarily apply for process \( Q \): \( Q \) might have actions not present in \( P \). In this particular case, our process and its decomposition will not perform exactly the same action. If a process \( P \) is to send a value \( V \), we would expect the decomposition of \( P \) to send the breakdown of value \( V \) as further detailed in Table 2. As a consequence, we define the compatibility function in Table 1 to embody this relation between the labelled transitions of processes.

Weak simulation is a class of simulation in which silent actions can be ignored, therefore \( Q \) must be able to perform the same action as \( P \) before and/or after a arbitrary amount of silent actions (including 0).

\[\text{Notation 3. } \text{We use } \xrightarrow{\ell^0} \text{ to specify 0 or more silent actions.}\]
Weak similarity is proven via a weak simulation by building a binary relation $R$ containing all pairs of the form $(P_i, Q_j)$ where $P_i$ and $Q_j$ represent $P$’s and respectively $Q$’s transitions and $Q_i$ weakly similar to $P_j$.

In the instance of our process, we came up with a specific relaxation of the definition of weak simulation which will have a couple of distinct characteristics:

First of all, $Q$ must be able to match the actions of $P$ only after an arbitrary amount of silent actions: $Q \xrightarrow{\tau} Q_i \Rightarrow \alpha \xrightarrow{} Q_j$ instead of $Q \xrightarrow{\tau} \Rightarrow \alpha \xrightarrow{} \tau \xrightarrow{} Q_j$.

Moreover, if $P$ executes action $\alpha$, then $Q$ should match it by performing an action $\beta$ which is compatible to $\alpha$ according to 2.3.

▶ Definition 2.2. Two processes $P$ and $Q$ are called weakly similar, written $P \subseteq Q$, if there exists a weak simulation $R$ such that $PRQ$ and the actions which they perform are compatible 2.3.

(1) For all $P_i$ and $\alpha$ with $P \xrightarrow{\alpha} P_i$, there exists $Q_j$ and $\beta$ such that $Q \xrightarrow{\tau} \beta \xrightarrow{} Q_j$ with $\text{Compat}(\alpha) = \beta$ and $P_i R Q_j$.

▶ Definition 2.3. Two labelled transitions $\alpha$ and $\beta$ are called to be compatible, denoted as $\text{Compat}(\alpha) = \beta$, if the following holds:

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$\text{Compat}(\alpha)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u!\langle V \rangle$</td>
<td>$u_i!\langle W \rangle$</td>
</tr>
<tr>
<td>$u?\langle V' \rangle$</td>
<td>$u_i?\langle V' \rangle$</td>
</tr>
<tr>
<td>$\tau$</td>
<td>$\tau$</td>
</tr>
</tbody>
</table>

Table 1 Compatibility function for labelled transitions.

NOTE: In the first row of the table, $W$ represents the breakdown of value $V$ further detailed in Table 2.

2.3 Program Decomposition

2.3.1 Session Types for HO

Session types represent a significant element when talking about parallel programming and process calculi as they can check whether a program is conforming to protocols and acting in a way it is expected to.

▶ Definition 2.4 (Session Types for HO [3]). Let us write $\diamond$ to denote the process type. The syntax of types for HO is defined as follows:

$$
U ::= C \rightarrow \diamond \mid C \rightarrow \diamond \\
C ::= S \mid \langle U \rangle \\
S ::= \text{end} \mid \langle !U \rangle; S \mid \langle ?U \rangle; S \mid \oplus \{l_i : S_i\}_{i \in I} \mid \& \{l_i : S_i\}_{i \in I} \mid \mu t. S \mid t
$$

Value types $U$ include $C \rightarrow \diamond$ and $C \rightarrow \diamond$, which denote shared and linear higher-order types, respectively. Shared channel types are denoted by $\langle S \rangle$ and $\langle U \rangle$.

Duality is a relation describing two end points of a communication link and intuitively it is obtained by exchanging $!$ by $?$ and vice versa. In this case, we have the output types and input types $\langle !U \rangle; S$, respectively $\langle ?U \rangle; S$, which are said to be duals of each-other. Output
type \( !\langle U \rangle; S \), is assigned to a name that first sends a value of type \( U \) and then continues interaction as the type described by \( S \). \textit{Input type} \( ?\langle U \rangle; S \), receives a value of type \( U \) and then continues interaction as the type described by \( S \). Duality is a key notion in session types as it assures that the interaction exhibits as expected and respects run-time protocols.

▶ Definition 2.5 (Minimal Session Types). The syntax of minimal session types for HO is defined as follows:

\[
\begin{align*}
U & ::= \tilde{C} \to \Diamond \mid \tilde{C} \to \Diamond \\
C & ::= M \mid \langle U \rangle \\
M & ::= \text{end} \mid !\langle \tilde{U} \rangle; \text{end} \mid ?(\tilde{U}); \text{end}
\end{align*}
\]

Minimal session types are a reduced set of session types for HO developed by following the definition of session types in “Characteristic bisimulation for higher-order session processes” [3]. Sequentially, we give the necessary properties developed by A. Arslanagić, J. A. Pérez, and E. Voogd in “Minimal Session Types” [1].

Value types \( U \) include \( \tilde{C} \to \Diamond \) and \( \tilde{C} \to \Diamond \), which denote \textit{shared} and \textit{linear} higher-order types, respectively. Shared channel types are denoted by \( \langle M \rangle \) and \( \langle U \rangle \).

\textit{Output type} \( !\langle \tilde{U} \rangle; \text{end} \) is assigned to a name that first sends a value of type \( U \) and then ends interaction whereas \textit{input type} \( ?(\tilde{U}); \text{end} \) receives a value of type \( U \) and then ends interaction.

▶ Definition 2.6 (Decomposing Session Types). Let \( S \) be a session type, \( U \) be a higher-order type, \( C \) be a name type, and \( \langle U \rangle \) be a shared type, all as in Def. 2.5. The type decomposition function \( G(\cdot) \) is defined as:

\[
\begin{align*}
G(!\langle U \rangle; S) &= \begin{cases} 
!G(U); \text{end} & \text{if } S = \text{end} \\
!G(U); \text{end}, G(S) & \text{otherwise}
\end{cases} \\
G(?\langle U \rangle; S) &= \begin{cases} 
?G(U); \text{end} & \text{if } S = \text{end} \\
?G(U); \text{end}, G(S) & \text{otherwise}
\end{cases} \\
G(\text{end}) &= \text{end} \\
G(C \to \Diamond) &= G(C) \to \Diamond \\
G(C \to \Diamond) &= G(C) \to \Diamond \\
G(\langle U \rangle) &= \langle G(U) \rangle \\
G(S_1, \ldots, S_n) &= G(S_1), \ldots, G(S_n)
\end{align*}
\]

Therefore we can mention that if a session type \( S \) contains \( k \) visible actions, the list \( G(S) \) will contain \( k \) minimal session types.

▷ Notation 4. \(|G(S)|\) is denoting the length of \( G(S) \).

2.3.2 Preliminary Definitions and Main Idea

The main idea of the decomposition of HO process \( P \) is developing a new process called \( D(P) \) typed using \textit{minimal session types}. By using a breakdown function on the inner component of \( P \), the decomposition will contain a series of processes with exactly three nested prefixes called trios [4].
Therefore, if $P$ contains $k$ nested actions in sequential order $D(P)$ will have $k$ trios running in parallel that must maintain the sequence of the prefixes in $P$.

Definition 2.7 (Degree of a Process and Value). Let $P$ be an HO process. The degree of $P$, denoted $|P|$, is inductively defined as follows:

$$|P| = \begin{cases} 
|V| + |Q| + 1 & \text{if } P = u_i!(V).Q \\
|Q| + 1 & \text{if } P = u_i!(y).Q \text{ or } P = u_i?(y).Q \\
|V| + 1 & \text{if } P = V u_i \\
|P'| & \text{if } P = (v s : S)P' \\
|Q| + |R| + 1 & \text{if } P = Q | R \\
1 & \text{if } P = y u_i \text{ or } P = 0
\end{cases}$$

The degree of a value $V$, denoted $|V|$, is defined as follows:

$$|V| = \begin{cases} 
|P| & \text{if } V = \lambda x : C^\omega . P \\
0 & \text{if } V = \lambda x : C^\omega . P \text{ or } V = y
\end{cases}$$

As a result of using a breakdown function we will generate multiple propagator names (or simply propagators), denoted with $c_k, c_{k+1}, \ldots$ that are meant to carry a tuple of variables denoted by $\tilde{x}$ and called the context of a trio. The propagators will trigger each other ending up in mimicking the sequential order present in the initial process, $P$.

Definition 2.8 (Name and Process Initialization). Let $\tilde{u} = (a, b, s, s', \ldots)$ be a finite tuple of names. We shall write $\text{init}(\tilde{u})$ to denote the tuple $(a_1, b_1, s_1, s'_1, \ldots)$. We will say that a process has been initialized if all of its names have some index.

Definition 2.9 (Decomposing Processes). Let $P$ be a closed HO process such that $\tilde{u} = \text{fn}(P)$. The decomposition of $P$, denoted $D(P)$, is defined as:

$$D(P) = (\nu \tilde{c})(\tilde{x}!)0 \mid B_k(\text{P} \sigma)$$

where: $k > 0; \tilde{c} = (c_k, \ldots, c_{k+|P|-1}); \sigma = \{\text{init}(\tilde{u})/\tilde{u}\}$ and $B_k(\cdot)$ as presented in Table 2.

Intuitively, the degree of the decomposition of a process will be larger than the degree of the process itself as a consequence of having multiple parallel sequences containing the propagator names $(c_k, c_{k+1}, \ldots)$.

Notation 5. $P_{\sigma}$ stands for $P$ where all $u$ with $u \in \text{fn}(P)$ are replaced by $\sigma(u)$.

Finally, in the table below the breakdown functions for processes and values are defined:
An exercise on program equivalence for message-passing processes

<table>
<thead>
<tr>
<th>$P$</th>
<th>$B^k_b(P)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u_1! (V).Q$</td>
<td>$c_k?(\overline{x}).u_1! (\overline{y}<em>{k+1} (V \sigma)).x</em>{k+1}(\overline{z})</td>
</tr>
<tr>
<td>$u_2? (y).Q$</td>
<td>$c_k?(\overline{x}).u_2?! (y).x_{k+1}(\overline{z})</td>
</tr>
<tr>
<td>$V_u_i$</td>
<td>$c_k?(\overline{x}).V^k_{k+1}(V) \tilde{m}$</td>
</tr>
<tr>
<td>$(\nu s : C)P'$</td>
<td>$(\nu \tilde{s} : G(C)) B^k_b(P' \sigma)$</td>
</tr>
<tr>
<td>$Q</td>
<td>R$</td>
</tr>
<tr>
<td>$0$</td>
<td>$c_k?(().0)$</td>
</tr>
<tr>
<td>$y$</td>
<td>$y$</td>
</tr>
<tr>
<td>$\lambda u : C^{\neg \neg}.P$</td>
<td>$\lambda \tilde{y} : G(C)^{\neg \neg}.(\nu \tilde{c}) (\overline{c_{k+1}(\overline{z})}</td>
</tr>
</tbody>
</table>

Table 2: The breakdown function for processes and values (core fragment) taken from [1].

## 3 An Exercise on Program Equivalence

In the present section we will be considering a process $P$ and it’s decomposition, $D(P)$, that will be analyzed for the objective of building a relation $R$ showcasing weak similarity between the two.

First and foremost the definition of $P$ is to be considered:

$$P = (u! (m).\nu y? (b).0 | \pi? (x).x? (true).0)$$

Process $P$ contains a parallel composition that at the beginning allows for the name $m$ to be sent across channel $u$. After synchronization, $m$ substitutes the placeholder $x$ and sends back a Boolean of the value $\text{true}$.

However, since $m$ is a name and HO lacks name-passing this is not possible. The workaround to this shortage is enacted by using an encoding of processes extracted from [1]. The encoding
is denoted as $\mathbf{[\cdot]}$ and will pack a name inside an abstraction and send it as a value.

**Definition 3.1.** The encoding of process $P$ is defined as follows:

$$[P] = (Q \mid R)$$

$$Q = u!(V).\overline{m}?(y).(\nu s)(y s \mid \pi!(\lambda b.0).0)$$

$$R = \pi?(y).(\nu s)(y s \mid \pi!(W).0)$$

$$V = \lambda z. z?(x).(x m)$$

$$W = \lambda x. x!(W').0 \text{ with } W' = \lambda z. z?(x).(x \text{ true})$$

Further, the decomposition $D([P])$ is to be considered:

$$D([P]) = (\nu c_1, \ldots, c_{19})\left(\overline{m}!(\cdot) \mid (c_1?().\overline{m}!!().\overline{m}!!() \mid B^2_1(Q) \mid B^{11}_1(R))\right)$$

$$B^2_1(Q) = \nu s_1().u!(V).\overline{m}!!().\overline{m}!!() \mid c_5?().\overline{m}??(y).\overline{m}!!(y) \mid$$

$$(\nu s_1)(c_9?().\overline{m}!!(y).\overline{m}!!() \mid c_7?().(y s_1) \mid c_8?().\overline{m}!!(\lambda b.0).\overline{m}!!() \mid c_{10}?())$$

$$B^{11}_1(R) = c_{11}?().\overline{m}??(y).\overline{m}!!(y) \mid$$

$$(\nu s_1)(c_{12}?().\overline{m}!!(y).\overline{m}!!() \mid c_{13}?().(y s_1) \mid c_{14}?().\overline{m}!!(V)^{15}(W).\overline{m}!!() \mid c_{19}?())$$

$$V^1(Q) = \lambda z_{21}.(\overline{m}!!() \mid c_3?().z_1?(x).\overline{m}!!(x) \mid c_4?().(x m_1))$$

$$V^6(\lambda b.0) = \lambda b_{11}.(\overline{m}!!() \mid c_9?)$$

$$V^{15}_x(W) = \lambda x_{11}.(\overline{m}!!() \mid c_{15}?().x_1!(V_x^{16}(W)).\overline{m}!!() \mid c_{18}?())$$

$$V^{16}_c(W) = \lambda z_{21}.(\overline{m}!!() \mid c_{16}?().z_1?(x).\overline{m}!!(x) \mid c_{17}?().(x \text{ true}))$$

We start by stating and proving the main contribution of the research work:

**Proposition 3.2.** Let $[P]$ be as in 3.1, we show that $[P] \sqsubseteq D([P])$ according to Definition 2.2.

**Proof.** The original course of action was constructing a tree starting with $[P]$ that features every action that the root, $[P]$, and all of its transitional states can exhibit. Sequentially, the same logic should apply in constructing a second tree displaying the decomposition of $[P]$ with the mention that in this case it is not mandatory to have every single transition that $D(\mathbf{[\cdot]})$ could evolve into.

The previously stated is a follow-up to the similarity definition that is asymmetric:

For a process $P$, to be similar to a process $Q$, $P$ needs to be able of mimicking all actions of $Q$ and of the transitions of $Q$, but not necessarily the other way around.

Attempting this approach, however, has proved inefficient and difficult to keep track of in respect to the high amount of transitional nodes present in the tree for $D(\mathbf{[\cdot]})$.

In that event, we proceed by taking the parallel components of $[P]$ separately to prove that $B^2_1(Q)$ is weakly similar to $Q$ and that $B^{11}_1(R)$ is weakly similar to $R$.

Later the interaction between those components will be taken in consideration in the interest of building the relation for the whole process.
An exercise on program equivalence for message-passing processes

To simplify the typing of \( B^2_0(Q) \) and \( B^{11}_1(R) \) the following notations have been defined:

\[
\begin{align*}
B^2_0(Q) &= c_2?().u_1!(\nu^0(Q)).\pi!() | c_5?().\tau!(y).\pi!(y) | (\nu s_1)(A_1 | A_2 | A_3 | A_4) \\
A_1 &= c_5?().\tau!(y).\pi!(y) | A_2 = c_7?().(y.s_1) | A_3 = c_8?().\tau!(Q^0_0(\lambda b.0)).\pi!(y) | A_4 = c_{10}?() \\
B^{11}_1(R) &= c_{11}?().\tau!(y).\pi!(y) | (\nu s_1)(B_1 | B_2 | B_3 | B_4) \\
B_1 &= c_{12}?().\tau!(y).\pi!(y) | B_2 = c_{13}?().(y.s_1) | B_3 = c_{14}?().\tau!(Q^1_0(\nu^1_0(W))).\pi!(y) | B_4 = c_{19}?()
\end{align*}
\]

In the context of later combining the relations of the parallel components of \( D([P_i]) \), we will assume that the inputs on channels \( c_2 \) and \( c_{11} \) have synchronized in order to trigger their corresponding breakdowns, \( B^2_0(Q) \) and \( B^{11}_1(R) \).

**Definition 3.3.** We define the new processes \( Q' \) and \( R' \) where \( B^2_0(Q) = c_2?().Q' \) and \( B^{11}_1(Q) = c_{11}?().R' \):

\[
\begin{align*}
Q' &= u_1!(\nu^0_1(Q)).\pi!() | c_5?().\tau!(y).\pi!(y) | (\nu s_1)(A_1 | A_2 | A_3 | A_4) \\
R' &= \tau!(y).\pi!(y) | (\nu s_1)(B_1 | B_2 | B_3 | B_4)
\end{align*}
\]

The transition tables for \( Q, Q', R \) and \( R' \) have been developed by using the labelled transition system for processes mentioned in 2.1.2.

**Example 3.4.** The labelled transition tree for \( Q \xrightarrow{\text{snd}} Q_1 \).

\[
\begin{array}{cccc}
\text{snd} & \tau!(y).\pi!(y) & (\nu s)(y.s) & \tau!(\lambda b.0.0) \\
\hline
\end{array}
\]

**Example 3.5.** The labelled transition tree for \( R'_1 \xrightarrow{\text{tau}} 0 \).

\[
\begin{array}{cccc}
\text{tau} & \tau!(y) & (\nu s_1)(e) & 0 \\
\text{snd} & \tau!(y) & (\nu s_1)(e) & 0 \\
\hline
\text{res} & (\nu s_1)(\nu z_1) & (\nu x_1)(e) & (\nu s_1)(\nu z_1) \\
\text{res} & (\nu s_1)(\nu z_1) & (\nu x_1)(e) & (\nu s_1)(\nu z_1) \\
\text{tau} & (\nu s_1)(\nu z_1) & (\nu x_1)(e) & (\nu s_1)(\nu z_1)
\end{array}
\]

**NOTE:** For rule TAU to be valid we made sure that \( c_{19}?() \geq \tau!() \) holds. Similarly, for rules RES we checked that \( s_1, z_1, x_1 \notin \text{fn}(c_{19}?()) \) and \( s_1, z_1, x_1 \notin \text{fn}(\tau!()) \) hold.
Table 3 The transition states of processes \( Q \) and \( R \).

\[
\begin{array}{|c|c|}
\hline
Q & u! (V), \tilde{\pi}?(y), (\nu s)(y s | \tilde{\pi}(\lambda b).0) \\
\hline
Q_1 & \tilde{\pi}?(y), (\nu s)(y s | \tilde{\pi}(\lambda b).0) \\
\hline
Q_{2s} & (\nu s)(V' s | \tilde{\pi}(\lambda b).0) \quad \sigma = \{W'/V'\} \\
\hline
Q_3 & (\nu s)(s? (x), (x true) | \tilde{\pi}(\lambda b).0) \\
\hline
Q_4 & (\nu s)(\nu b)((\lambda b).0)true) \\
\hline
0 & (\nu s)(\nu b)(0) \quad (\nu n)0 \equiv 0 \\
\hline
\end{array}
\]

Table 4 The transition states of processes \( Q' \) and \( R' \).

\[
\begin{array}{|c|c|}
\hline
Q' & u_1! O^2_s(V), \tilde{\pi}!(\cdot), c_s?(\cdot), \tilde{\pi}!(y), \tilde{\pi}!(\cdot y) | (\nu s_1)(A_1 | A_2 | A_3 | A_4) \\
\hline
Q'_1 & \tilde{\pi}!(\cdot) | c_s?(), \tilde{\pi}!(\cdot y), \tilde{\pi}!(\cdot y) | (\nu s_1)(A_1 | A_2 | A_3 | A_4) \\
\hline
Q'_2 & \tilde{\pi}!(y), \tilde{\pi}!(\cdot y) | (\nu s_1)(A_1 | A_2 | A_3 | A_4) \\
\hline
Q'_{1s} & \tilde{\pi}!(V') | (\nu s_1)(A_1 | A_2 | A_3 | A_4) \\
\hline
Q'_{2s} & (\nu s_1)(\nu bn(V'))(V' s_1 | A_1 | A_4) \quad \sigma = \{V^{16}_s(W')/V'\} \\
\hline
Q'_3 & (\nu s_1)(\nu x_1)(\nu z_1)(\tilde{\pi}!(\cdot)) | c_16?(), s_1?(x), \tilde{\pi}!(x) | c_17?((x), (x true) | A_4) \\
\hline
Q'_{4s} & (\nu s_1)(\nu x_1)(\nu z_1)(\tilde{\pi}!(x)) | c_17?((x), (x true) | A_4) \\
\hline
0 & (\nu s_1)(\nu x_1)(\nu z_1)(0) \quad (\nu n)0 \equiv 0 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
R' & \tilde{\pi}!(\cdot y), \tilde{\pi}!(\cdot y) | (\nu s_1)(B_1 | B_2 | B_3 | B_4) \\
\hline
R'_{1s} & \tilde{\pi}!(\cdot y) | (\nu s_1)(B_1 | B_2 | B_3 | B_4) \\
\hline
R'_{2s} & (\nu s_1)(\nu bn(V'))(V' s_1 | \tilde{\pi}!(W^{15}_s(W)).\tilde{\pi}!(\cdot) | B_4) \quad \sigma = \{V^{2}_s(V)/V'\} \\
\hline
R'_{3s} & (\nu s_1)(\nu z_1)(\nu x_1)(\tilde{\pi}!(\cdot)) | c_3?(), s_1?((x), \tilde{\pi}!(x) | c_4?((x), (x m_1) | \tilde{\pi}!(W^{15}_s(W)).\tilde{\pi}!(\cdot) | B_4) \\
\hline
R'_{4s} & (\nu s_1)(\nu z_1)(\nu x_1)(\tilde{\pi}!(\cdot)) | c_4?((x), (x m_1) | \tilde{\pi}!(W^{15}_s(W)).\tilde{\pi}!(\cdot) | B_4) \\
\hline
R'_{5s} & (\nu s_1)(\nu x_1)(\nu z_1)(\tilde{\pi}!(\cdot)) | c_5?((x), (x m_1) | \tilde{\pi}!(W^{15}_s(W)).\tilde{\pi}!(\cdot) | B_4) \\
\hline
R'_{6s} & (\nu s_1)(\nu x_1)(\nu z_1)(\tilde{\pi}!(\cdot)) | c_6?((x), (x m_1) | \tilde{\pi}!(W^{15}_s(W)).\tilde{\pi}!(\cdot) | B_4) \\
\hline
R'_{7s} & (\nu s_1)(\nu x_1)(\nu z_1)(\tilde{\pi}!(\cdot)) | c_7?((x), (x m_1) | \tilde{\pi}!(W^{15}_s(W)).\tilde{\pi}!(\cdot) | B_4) \\
\hline
R'_{8s} & (\nu s_1)(\nu x_1)(\nu z_1)(\tilde{\pi}!(\cdot)) | c_8?((x), (x m_1) | \tilde{\pi}!(W^{15}_s(W)).\tilde{\pi}!(\cdot) | B_4) \\
\hline
0 & (\nu s_1)(\nu x_1)(\nu z_1)(0) \quad (\nu n)0 \equiv 0 \\
\hline
\end{array}
\]

In the occasions when an input is to be received, \( V' \) is used to denote an arbitrary value.
We recall that $P_\sigma$ stands for $P$ where all $u$ with $u \in \mathcal{F}_n(P)$ are replaced by $\sigma(u)$. In order to facilitate the continuation of internal actions, any unspecified input $V'$ will be substituted by an appropriate value. As an exemplification, $\sigma = \{V_1^{16}(W')\}$ is a substitution where $V_1^{16}(W')$ will take the place of $V'$ and continue interaction until the process reaches 0 or a state congruent with 0.

We proceed by providing trees as a visual representation of process equivalence between $Q$ and $Q'$.

In order to prove that $Q'$ is weakly similar to $Q$, a relation $\mathcal{R}$ must be found containing the pair $(Q,Q')$. Therefore, the first step in building the relation is adding $(Q,Q')$ to $\mathcal{R}$.

$$\mathcal{R} = \{(Q,Q')\}$$

For $\mathcal{R}$ to be a weak simulation, all derivatives of $Q$ should appear in the first component of each pair. The main idea is listing all of $Q$’s transitional states and finding their corresponding state from the transitions of $Q$.

In this instance, if a process can receive or send a value $V$, it’s corresponding pair should be able to receive, respectively send, the breakdown of value $V$ after an arbitrary amount of $\tau$ actions.
Since we are following definition 2.2, the labelled transitions have been checked to respect the properties of the compatibility function mentioned in Table 1.

\[ Q \xrightarrow{u_1(V)} Q_1 \]
\[ Q_1 \xrightarrow{m_1(V')} Q_{2\sigma} \]
\[ Q_{2\sigma} \xrightarrow{\tau_{\sigma} = (V')} Q_3 \]
\[ Q_3 \xrightarrow{\tau} Q_4 \]
\[ Q_4 \xrightarrow{\tau} 0 \]
\[ Q' \xrightarrow{u_1(V')} Q'_1 \]
\[ Q'_1 \xrightarrow{m_1(V')} Q'_{1\sigma} \]
\[ Q'_{3\sigma} \xrightarrow{\tau = (V'^{16})} Q'_{7} \]
\[ Q'_5 \xrightarrow{\tau} Q'_8 \]
\[ Q'_8 \xrightarrow{4\tau} 0 \]

\[ R = \{ (Q, Q'); (Q_1, Q'_1); (Q_{2\sigma}, Q'_{3\sigma}); (Q_3, Q'_5); (Q_4, Q'_8); (0, 0) \} \]

The same procedure is applied to \( R \) and \( R' \):

**Figure 3** Tree representing the transitional states of process \( R \) and \( R' \)
An exercise on program equivalence for message-passing processes

Now that the equivalence relations between the parallel components of $[P]$ and their breakdowns have been built, the following step would be using them in developing a weak simulation between $[P]$ and $D([P])$.

We start by exploring every transitional state of process $[P]$ with the mention that we will stop expanding the tree once a node will reach a state that has been previously used in the weak simulations created.

For instance, state $P_{12}$ evolves into $R$, that further transitions into $R_{1} \sigma \rightarrow R_{2} \rightarrow R_{3} \rightarrow R_{4} \rightarrow 0$.

In order to properly build the final relation $R$ we need to ensure that it will contain every transitional state of $P$, this however, not being mandatory for the transition states of $D([P])$.

Since we know that $R \rightarrow R_{1} \sigma \rightarrow R_{2} \rightarrow R_{3} \rightarrow R_{4} \rightarrow 0$ and $Q \rightarrow Q_{1} \rightarrow Q_{2} \sigma \rightarrow Q_{3} \rightarrow Q_{4} \rightarrow 0$, we can conclude that the final state of process $P$ will be $0$. Considering relations (12) and (6) we can add the corresponding pairs $(0, R'_{12})$ and $(0, 0)$ to $R$.

Building Table 5 containing the transition states of process $P$ has proven less complex once we made use of the information already developed for each parallel component.

Having available every possible evolution states of processes $Q$ and $R$ we have combined them and followed their interaction when positioned in parallel.

Two interactions between the components have been observed, the first one being a synchronization at the very beginning along channel $u$, where the value $V$ is passed along. Sequentially, another synchronization takes place further down in the branching from state $P_{12}$, where value $W'$ is passed along channel $m$.

The tree has a slightly symmetric aspect as we follow the development of the states where no synchronization takes place, respectively on the furthest left and right branches.

\[
\begin{align*}
R & \xrightarrow{\tau_{(v)}} R_{1} \\
R_{1} & \xrightarrow{\tau_{(v)}} R_{2} \\
R_{2} & \xrightarrow{\tau_{}} R_{3} \\
R_{3} & \xrightarrow{\tau_{}} R_{4} \\
R_{4} & \xrightarrow{m_{1}(W')} 0
\end{align*}
\]
Figure 4 Tree representing the transitional states of process $[P]$ as nodes and the respective possible actions of the processes including input, output and silent actions.
An exercise on program equivalence for message-passing processes

<table>
<thead>
<tr>
<th>$[P]$</th>
<th>$Q \mid R$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>$Q_1 \mid R$</td>
</tr>
<tr>
<td>$P_2$</td>
<td>$Q \mid R_{1\sigma}$</td>
</tr>
<tr>
<td>$P_3$</td>
<td>$Q_1 \mid R_{1\sigma}$</td>
</tr>
<tr>
<td>$P_{4\sigma}$</td>
<td>$Q_{2\sigma} \mid R_{1\sigma}$</td>
</tr>
<tr>
<td>$P_5$</td>
<td>$Q_1 \mid R_2$</td>
</tr>
<tr>
<td>$P_6$</td>
<td>$Q_3 \mid R_{1\sigma}$</td>
</tr>
<tr>
<td>$P_7$</td>
<td>$Q_{2\sigma} \mid R_2$</td>
</tr>
<tr>
<td>$P_8$</td>
<td>$Q_1 \mid R_3$</td>
</tr>
<tr>
<td>$P_9$</td>
<td>$Q_4 \mid R_{1\sigma}$</td>
</tr>
<tr>
<td>$P_{10}$</td>
<td>$Q_3 \mid R_2$</td>
</tr>
<tr>
<td>$P_{11\sigma}$</td>
<td>$Q_{2\sigma} \mid R_3$</td>
</tr>
<tr>
<td>$P_{12}$</td>
<td>$Q_4 \mid R_4$</td>
</tr>
<tr>
<td>$P_{13}$</td>
<td>$Q_4 \mid R_2$</td>
</tr>
<tr>
<td>$P_{14}$</td>
<td>$Q_3 \mid R_4$</td>
</tr>
<tr>
<td>$P_{15}$</td>
<td>$Q_{2\sigma} \mid R_4$</td>
</tr>
<tr>
<td>$P_{16}$</td>
<td>$Q_4 \mid R_3$</td>
</tr>
<tr>
<td>$P_{17}$</td>
<td>$Q_3 \mid R_4$</td>
</tr>
<tr>
<td>$P_{18}$</td>
<td>$Q_4 \mid R_4$</td>
</tr>
<tr>
<td>$P_{19}$</td>
<td>$Q_{2\sigma} \mid R$</td>
</tr>
<tr>
<td>$P_{20}$</td>
<td>$Q_3 \mid R$</td>
</tr>
<tr>
<td>$P_{21}$</td>
<td>$Q_4 \mid R$</td>
</tr>
<tr>
<td>$P_{22}$</td>
<td>$Q \mid R_2$</td>
</tr>
<tr>
<td>$P_{23}$</td>
<td>$Q \mid R_3$</td>
</tr>
<tr>
<td>$P_{24}$</td>
<td>$Q \mid R_4$</td>
</tr>
</tbody>
</table>

Table 5 The transition states of process $[P]$.  

Moreover, since we know exactly which states of $B_2^2(J) \cap B_1^{11}(R)$ can mimic the ones of $Q$, respectively $R$, we can also easily find out which states of $D([P])$ can mimic the ones of $[P]$. This is represented in Table 6, containing states $P_1\ldots P_{18}$ of $D([P])$ as the mimicking pairs for $P_1\ldots P_{18}$ of $[P]$. For example, let’s say we are aiming to find a transition state of $D([P])$ which is able of matching the actions of $P_5$. We first take a look at Table 5 to see that $P_5$ contains the parallel component consisting of states $Q_1$ and $R_2$.

We further proceed to find the pairs of $Q_1$ and $R_2$ from the weak simulations created for
processes $Q$ in (6), respectively, $R$ in (12). Seeing that the match for $Q_1$ is $Q'_1$ and the one for $R_2$ is $R'_5$, we finalize by putting them in parallel and creating state $P'_5 = (Q'_1 \mid R'_5)$.

\[
    \mathcal{D}([P]) \equiv (\nu c_1, \ldots, c_{19}) (\tau_T() \mid (c_1?().\tau_T().\tau_T()) \mid B_5^2(Q) \mid B_5^2(R))
\]

<table>
<thead>
<tr>
<th>$P'_{i}$</th>
<th>$Q'_i \mid R'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P'_1$</td>
<td>$Q'_1 \mid R'$</td>
</tr>
<tr>
<td>$P'_2$</td>
<td>$Q' \mid R'_{1\sigma}$</td>
</tr>
<tr>
<td>$P'_3$</td>
<td>$Q'<em>1 \mid R'</em>{1\sigma}$</td>
</tr>
<tr>
<td>$P'_4$</td>
<td>$Q'<em>{3\sigma} \mid R'</em>{1\sigma}$</td>
</tr>
<tr>
<td>$P'_5$</td>
<td>$Q'_1 \mid R'_5$</td>
</tr>
<tr>
<td>$P'_6$</td>
<td>$Q' \mid R'_{1\sigma}$</td>
</tr>
<tr>
<td>$P'_7$</td>
<td>$Q'<em>{3\sigma} \mid R'</em>{1\sigma}$</td>
</tr>
<tr>
<td>$P'_8$</td>
<td>$Q'_1 \mid R'_6$</td>
</tr>
<tr>
<td>$P'_9$</td>
<td>$Q'<em>8 \mid R'</em>{1\sigma}$</td>
</tr>
<tr>
<td>$P'_{10}$</td>
<td>$Q'_7 \mid R'_5$</td>
</tr>
<tr>
<td>$P'_{11}$</td>
<td>$Q'_{3\sigma} \mid R'_6$</td>
</tr>
<tr>
<td>$P'_{12}$</td>
<td>$Q'_1 \mid R'_7$</td>
</tr>
<tr>
<td>$P'_{13}$</td>
<td>$Q'<em>8 \mid R'</em>{1\sigma}$</td>
</tr>
<tr>
<td>$P'_{14}$</td>
<td>$Q'_7 \mid R'_6$</td>
</tr>
<tr>
<td>$P'_{15}$</td>
<td>$Q'_{3\sigma} \mid R'_7$</td>
</tr>
<tr>
<td>$P'_{16}$</td>
<td>$Q'_8 \mid R'_6$</td>
</tr>
<tr>
<td>$P'_{17}$</td>
<td>$Q'_7 \mid R'_5$</td>
</tr>
<tr>
<td>$P'_{18}$</td>
<td>$Q'_8 \mid R'_7$</td>
</tr>
<tr>
<td>$P'_{19}$</td>
<td>$Q'_{3\sigma} \mid R'$</td>
</tr>
<tr>
<td>$P'_{20}$</td>
<td>$Q'_7 \mid R'$</td>
</tr>
<tr>
<td>$P'_{21}$</td>
<td>$Q'_8 \mid R'$</td>
</tr>
<tr>
<td>$P'_{22}$</td>
<td>$Q' \mid R'_{1\sigma}$</td>
</tr>
<tr>
<td>$P'_{23}$</td>
<td>$Q' \mid R'_{1\sigma}$</td>
</tr>
<tr>
<td>$P'_{24}$</td>
<td>$Q' \mid R'_7$</td>
</tr>
</tbody>
</table>

**Table 6** The transition states of process $\mathcal{D}([P])$.

The transition tables for $[P]$ and $\mathcal{D}([P])$ have been developed by using the labelled transition system for processes mentioned in 2.1.2. Finally, based on all of the information formerly found, the weak simulation for the whole process and its decomposition is put together:
Starting by adding the pair \( ([P], D([P])) \) to the relation, we observe that the next state \([P]\) is going to evolve into is \(P_1\). Similarly, \(D([P])\) will mimic this action by transitioning into \(P'_1\) after a couple of silent transitions. Those are triggered by the synchronization of the propagator names \(\gamma_1(); \gamma_2(); \gamma_1()\) with their corresponding duals. As a result, we add \((P_1, P'_1)\) to \(\mathcal{R}\) and continue this procedure up to the moment \([P]\) cannot perform any further actions:

\[
\mathcal{R} = \{([P], D([P])); (P_1, P'_1); (P_2, P'_2); (P_3, P'_3); (P_4, P'_4); (P_5, P'_5); (P_6, P'_6); (P_7, P'_7); (P_8, P'_8); (P_9, P'_9); (P_{10}, P'_{10}); (P_{11}, P'_{11}); (P_{12}, P'_{12}); (P_{13}, P'_{13}); (P_{14}, P'_{14}); (P_{15}, P'_{15}); (P_{16}, P'_{16}); (P_{17}, P'_{17}); (P_{18}, P'_{18}); (P_{19}, P'_{19}); (P_{20}, P'_{20}); (P_{21}, P'_{21}); (P_{22}, P'_{22}); (P_{23}, P'_{23}); (P_{24}, P'_{24}); (R, R'); (R_{1a}, R'_{1a}); (R_2, R'_2); (R_3, R'_3); (R_4, R'_4); (Q, Q'); (Q_1, Q'_1); (Q_{2a}, Q'_{3b}); (Q_3, Q'_7); (Q_4, Q'_6); (0, R'_{12}); (0, 0) \}
\]

4 Conclusion

In conclusion, my research has expanded on the work done by the fundamental computing group, plunging into the challenging subject of program equivalence. The main motivation behind choosing this topic was investigating the non-deterministic behaviour of processes and their interactions in order to better my understanding of concurrency theory: an ongoing and pressing matter in my field of study.

During the development of the present paper many challenges have arisen due to the complexity of dealing with the interaction of parallel processes and the fast paced branching that occurs when taking into account every possible steps of the interactions.

Having successfully developed a weak simulation between a process, \([P]\), and it’s decomposition, \(D([P])\), it is important to mention the asymmetry in this relation. Some future work could include creating a symmetric relation called weak bi-simulation between the two processes. The mentioned expansion of the formula could be done by improving the generated proof and demonstrating that \(D([P])\) is also weakly similar to \([P]\), whether that be possible or not.

References


