BACHELOR PROJECT THESIS

FINAL YEAR PROJECT

Relating Specifications of Compensations and Dynamic Update

Ana Roman
S2763753

Supervised by:
Dr. Jorge A. Pérez
Prof. Gerard Renardel de Lavalette

December 5, 2017
University of Groningen

Abstract

The present paper is a report of the research done during the final project of the Computing Science bachelor. Triggered by recent results by the Fundamental Computing group, I investigated the formal principles of modern programming languages with characteristics such as failure handling, self-adaptation and dynamic update, which are essential in programming business-oriented and service-oriented systems in the modern world. The research done involved exploring a wide variety of design options related to system transactions and their associated compensation actions (the actions that should take place in case a transaction fails). Various models have been proposed to include compensations, based on different formal approaches and abstract languages.
The main contribution of my work is a new formal translation between two formal languages for concurrency: one is Webπ, a calculus for long-running transactions; the other is the calculus of adaptable processes, a formal model for dynamic update [DPP15]. This report presents the formal definition of the translation, and analyses it via examples.

1 Introduction

Along with the evolution of computers, the scenario of business computing systems changed with the introduction of cross-entity computer interactions. In the past, computer systems did not interact with another's. Therefore, the main concern of these systems was to keep data safe and consistent, to process it as efficiently as possible, and to deal with concurrency issues if the system was accessed at the same time from several points (inside or outside the system). The concept of *transaction* was introduced in this situation to guarantee that a change brought to the data is either completed or undetectable [MMLI06].
In the modern day scenario, systems are not only concerned with the processing or storing of data, but become themselves the actors in real-life actions. This advancement rendered the traditional transaction mechanisms obsolete and insufficient in order to deal with the modern, very complex and longer multi-party transactions [MMLI06].
Modern software applications are built using long-running transactions (for short, LRTs). LRTs are a specific type of transactions. They are computing activities which extend in time and may involve resources. A particularly interesting feature of LRT management is handling partial or total failures. In this matter, the mechanisms that are meant to bring the LRT back to a correct state need to be explicitly programmed.
The concept of *compensations* has been introduced as an alternative, providing the possibility of executing a pre-defined, alternative behavior of a system in case a transaction fails. Therefore, rather than waiting for the whole transaction to complete, any completed part can be considered successful, taking into account the possibility that if some other system fails later, the affected parts can be compensated. Compensations do not necessarily mean reversing the actions entirely; rather, they specify an alternative behavior that should be executed upon failure.
When defining the behavior of a system, formal models can be used in order to describe the processes that take place. Formal models can be described through formal programming languages, by representing the states of the model, the behavior of LRTs and their compensations. My research involved the study of a number of these formal languages:

- π-Calculus
- Adaptable Processes
- Compensable Processes

and the translations between them. The task was to find a proper language which could best fit the language of Compensable Processes, in order to find a simple, elegant translation between the two. As source language in my study, I have taken into consideration twelve different languages out of which I have chosen Webπ as the best fit for Compensable Processes. Webπ is an extension of the asynchronous π-Calculus [MMLI06], with a special process for handling transactions.

1.1 Encoding

An encoding is a function that compiles a program in a source language into a program in a target language. In this way, translations can be seen as formal compilers that preserve the language's properties. An encoding is based on a translation, for which at least two correctness criteria must hold: operational correspondence and compositionality.
Operational correspondence is obtained when the following holds: if, in one source language, a process P can evolve in a certain way to reach the process P', then this evolution should somehow be mimicked in its translation in the target language. In other words, the two languages should be able to express similar operations. More formally, this property can be expressed in the following way: Let $[\![.]\!] : L_1 \rightarrow L_2$ be the encoding function between two languages.

$$\begin{array}{lll} \text{IF} & L_1: & P \xrightarrow{\alpha} P' \\ \text{THEN} & L_2: & [\![P]\!] \xrightarrow{\alpha} [\![P']\!] \end{array}$$

This example says the following: if, in language L_1, process P can evolve and become P' through a step (denoted α), the same process encoded in another language L_2 should be able to evolve (through the same step) into P' in the respective language. For the simplicity of this example, I chose the same steps denoted α in both of the languages, meaning that in both languages exactly the same step is being followed.
Another correctness criteria that has to hold in order for a translation to form a rigorous encoding is the principle of *compositionality*. The principle states that the translation of complex expressions is determined by the translation of its constituent expressions and the rules used to combine them. This can be seen in the way systems react to stimuli, by defining the behavior of the system based on the behavior of its independent components [1].
The motivation for finding a translation between two different languages has multiple important aspects, such as the following:

- A translation can connect specifications at different levels of abstraction.
- Given the ample linguistic diversity of the constructs of Webπ, understanding how they can be implemented as compensable processes could result in simpler models.
- The translation can allow the transfer of techniques, theories and insights from one language to the other.
- A new category of systems can be implemented using Compensable Processes, since Webπ deals with Web Services.

The project required developing new encodings of languages with compensations into run-time update. The intended translation has to consider the same target language as it was in prior research [DPP15], but it will expand on different target languages, less general with the purpose of finding simpler and more compact translations.

[1] `https://en.wikipedia.org/wiki/Principle_of_compositionality`

As contributions, my research has brought the following: An encoding that will allow the translation of models written in Webπ to be translated into Compensable Processes. Along with the encoding, I will present a compelling example to prove the correctness of the translation.

1.2 The structure of the paper

The paper is organized in the following way: Section §2 describes the background research that had to be done in order to understand the new concepts for the project. Section §3 describes the methodology used for the task of finding a new translation. Firstly, it presents the languages that were analyzed, then the best candidate for the translation, and continues with the necessary explanations that were required in order to design the new translation. Section §4 illustrates the correctness of the translation through an example, and §5 concludes and suggests possible ideas for future work.

2 Prelimiaries

2.1 Reactive systems

My study is based upon reactive systems and the notions used to describe them. Reactive systems are systems that react to stimuli from their environment. They are parallel systems, in which a key role is played by communication and interaction with their computing environment [AILS07].
Unlike in the setting of sequential programs, where the development of correct programs can be done with less recourse to formalisms, it is a recognized fact that the behavior of parallel and concurrent programs can be difficult to analyze and understand. The analysis requires a careful consideration of issues related to the interactions among the components, and as a result, CCS, along with Labeled Transition Systems provide the means for doing that.

2.2 CCS

The formal languages that I have studied are based on Milner's Calculus of Communicating Systems [AILS07]. CCS is a process calculus used to model communication between participants, with the use of primitives for describing parallel composition, choice between action and scope restriction. CCS can be used to describe and analyze any collection of interacting processes; as such, it gives the foundation for the development of (semi)automatic tools that support various formal methods for validation and verification, and that can be applied to the analysis of complex computing systems.
The syntax of CCS is the following:
$\pi ::=$

- (**nil**) $\mathbf{0}$
- (**input**) a
- (**output**) $\overline{a}$

$P ::=$

- (**nil**) $\mathbf{0}$
- (**prefix**) $\pi.P$
- (**process identifier**) A

- (**choice**) $\sum_{i \in I} P_i$
- (**parallel composition**) $P_1 | P_2$
- (**renaming**) $P[f]$
- (**restriction**) $P_1 \backslash \pi$

Where A is a process identifier from the countably infinite collection of process names (which ensures that we never run out of processes). π is an action, and the general rule for action prefixing says the following: if P is a process and π is a prefix, then $\pi.P$ is a process; this process begins by performing the prefix π and behaves like P thereafter. I is a possibly infinite index set. f is a relabeling function [AILS07].
In CCS, (complex) processes can be given names which are used to define process descriptions. A small example of a CCS process is the following:

$$Clock \stackrel{\text{def}}{=} tick.tock.0$$

The process defined above represents a simple process called Clock, which takes two inputs, *tick* and *tock* and then is stopped - it cannot proceed further into its computation.
The introduction of names allows us to give recursive definitions of process behaviors. Thus, we could change the example mentioned above the following way:

$$\begin{aligned} Clock &\stackrel{\text{def}}{=} tick.tock.Clock \\ &= tick.tock.tick.tock.Clock \\ &= tick.tock.tick.tock.tick.tock.Clock \\ &= tick.tock.....tock.Clock \qquad \text{(n-times)} \end{aligned}$$

A Labeled Transition System (LTS) is a model for giving semantics to process expressions, in which processes are represented by vertices of edge-labeled directed graphs, and in which a change of process state caused by an action is understood as moving along an edge, labeled by the action name, that goes out of that state. Therefore, a LTS consists of a set of states, a state of labels and a transition relation. The general form of a transition is the following: $P \xrightarrow{\alpha} P'$, where process P changes its state and becomes P' after performing action α [AILS07].

2.3 Adaptable and Compensable processes

This section is a short presentation of the calculus of Adaptable and Compensable Processes. The section describes already existing work done by the Fundamental Computing group, as it is illustrated in the paper "On Compensation Primitives as Adaptable Processes" [DPP15]. These languages establish the foundation of my research. The properties of each language will be described, focusing on the most important ones which were required for the development of the new translation.

2.3.1 Adaptable Processes

The calculus of *adaptable processes* was introduced as a variant of CCS, extended with two specific constructs. These constructs help represent the dynamic update of active communicating processes.
The syntax of adaptable processes is the following
The syntax of the calculus of adaptable processes is defined by prefixes $\pi.\pi'...$ and processes $P, Q...$:
$\pi ::=$

- (**input**) a

- (**output**) $\overline{a}$
- (**update prefix**) $l\{(X).Q\}$

where the update prefix may contain zero or more occurrences of *process variable*X.
$P, Q ::=$

- (**nil**) **0**
- (**located processes**) $l[P]$
- (**prefix or sequentiality**) $\pi.P$
- (**replication**) $!P$
- (**parallel composition**) $P|Q$
- (**restriction**) X

The two new constructs introduced in the calculus of Adaptable Processes are the following:

1. A located process, denoted $l[P]$, represents a process P which resides in a location called l.
   Locations are $transparent$: the behavior of $l[P]$ is the same as the behavior of P.
   Locations can also be arbitrarily $nested$, which allows to organize process descriptions into meaningful hierarchical structures.

2. An update prefix $l\{(X).Q\}$—where X is a process variable that occurs zero or more times in Q—denotes an adaptation mechanism for processes at location l.

The rest of the constructs are the usual CCS constructs for inaction, prefix (sequentiality), replication (when the process P is replicated continuously), parallel composition (processes P and Q are taking place at the same time) and restriction (cannot be accessed). The semantics of adaptable processes is given by a reduction semantics, denoted $\rightarrow$. Instead of giving the full semantics, we can look at the following example of a reduction step:
Example 3.1.1 Consider the following reduction:

$$C_1\big[l[P]\big] \mid C_2[l\{(X).Q\}.R] \rightarrow C_1[Q\{P/X\}] \mid C_2[R]$$

Here, C_1 and C_2 denote contexts: arbitrary, nested locations. The adaptation mechanism moves to the place where $l[P]$ resides and does a dynamic update there, as represented by the substitution $Q\{P/X\}$. Therefore, adaptation is a form of higher-order process communication [DPP15].

2.3.2 Compensable Processes

The core process language that is treated in my research is the one of compensable processes, due to its characteristics that allow us to capture compensation handling. The calculi comes as an extension of π-calculus, and it provides primitives that can capture static and dynamic recovery.
The syntax of Compensable Processes (CP)
Compensable processes extends CCS with constructs for transactions, protected blocks and compensation updates:
$P, Q ::=$

- (**nil**) **0**
- (**input or output**) $\pi.P$
- (**replication**) $!P$
- (**parallel composition**) $P|Q$

- (**transactions**) $t[P, Q]$, P, Q are processes
- (**protected blocks**) $\langle Q \rangle$, cannot be affected by abortion signals
- (**a process variable**) X
- (**compensation updates**)$\texttt{inst}\ \lfloor \lambda X.Q \rfloor$

The calculi makes use of the following constructs:

- *Transaction scopes (or transactions)*, denoted $t[P, Q]$, where t is a name, P, Q are both processes. P is the default behavior of the transaction, and Q is the compensation to be installed in case the transaction fails or an abortion signal arrives along name t. The abortion signals will lead to the discarding of the default behavior and the execution of Q.
  Transactions can be nested, so the process P may contain other transactions.
- *Protected blocks*, denoted $\langle Q \rangle$, for some process Q. These constructs protect the process from abortion signals. The protected blocks are transparent, meaning that $\langle Q \rangle$ and Q have the same behavior, but $\langle Q \rangle$ cannot be affected by abortion signals. Protected blocks are meant to prevent abortions after a compensation.
- *Compensation updates*, denoted $\texttt{inst}\ \lfloor \lambda X.Q \rfloor.P$ where P, Q are processes and X is a process variable that occurs zero or more times in Q.

In order to illustrate the semantics, we present some examples.
Example 3.2.1: Below, t is a location and P, Q are processes:

$$t[P, Q] | \overline{t} \rightarrow \langle Q \rangle$$

The example illustrates the situation where the transaction t will synchronize with $\overline{t}$; the transaction will be aborted and the compensating behavior Q is installed. At the same time, the example shows that the behavior Q will be protected and immune to external errors, due to the protected blocks.
Considering the case of the following nested transaction: $t_1[t_2[P_2, Q_2] | \overline{t_2}.R_1, Q_1]$, the semantics of compensations can partially preserve part of the behavior by using *extraction functions*, denoted $\texttt{extr()}$. For this example, we will have the following:

$$t_1[t_2[P_2, Q_2] | \overline{t_2}.R_1, Q_1] \rightarrow t_1[\langle Q \rangle | \texttt{extr}(P_2) | R_1, Q_1]$$

In this case, t_2 is aborted and the compensation behavior Q_2 is preserved, along with part of the behavior of P_2. The extraction function preserves at least all the protected blocks in P_2 but it can also contain other processes. The extraction function is discussed below more in depth.
The `extr()` function
For the extraction function, three variants are considered that define semantics for compensations. They mostly concern protected blocks and transactions. Given a process P, we have the following:

- $\texttt{extr}_D(P)$ denotes discarding semantics. It preserves only protected blocks in P, and discards other processes or transactions. It has the following notation: $\xrightarrow{\tau}_D$.
- $\texttt{extr}_P(P)$ denotes preserving semantics. It preserves protected blocks and top level transactions in P, discarding other processes.It has the following notation: $\xrightarrow{\tau}_P$.
- $\texttt{extr}_A(P)$ denotes aborting semantics. It preserves protected blocks and nested transactions in P, also including the compensation activities and discarding other processes. It has the following notation: $\xrightarrow{\tau}_A$.

The three semantics imply different levels of protection. Let us consider the following examples, by looking at the process $P = t[t_1[P_1, Q_1] \mid t_2[\langle P_2\rangle, Q_2] \mid R \mid \langle P_3\rangle, Q_5]$:

Discarding semantics:	$\bar{t} \mid P \xrightarrow{\tau}_D$	$\langle P_3\rangle \mid \langle Q_5\rangle$; only protected blocks are preserved.
Preserving semantics:	$\bar{t} \mid P \xrightarrow{\tau}_P$	$\langle P_3\rangle \mid \langle Q_5\rangle \mid t_1[P_1, Q_1] \mid t_2[\langle P_2\rangle, Q_2]$; preserves protected blocks and top level transactions.
Aborting semantics:	$\bar{t} \mid P \xrightarrow{\tau}_A$	$\langle P_3\rangle \mid \langle Q_5\rangle \mid \langle P_2\rangle \mid \langle Q_1\rangle \mid \langle Q_2\rangle$ preserves protected blocks, nested transactions and compensation activities.

3 Methodology

The objective of the study was to find a formal language different from CP for which the properties of operational correspondence and compositionality would hold most. Therefore, part of the research was to closely study 12 other formal languages and their syntax. This part of my research was based on the paper written by Colombo and Pace: "A Compensating Transaction Example in Twelve Notations" [CP11]. In their work, they review a number of languages and notations which can handle compensations by going through their syntax and semantics, encoding the same example into all of these notations, and discussing the strengths and disadvantages of each language.

3.1 Analyzing the languages

In the paper, the notations are organized depending on whether they assume a centralized collaboration between processes, orchestration approaches, or a decentralized collaboration, named choreography approaches. These two approaches are then compared, and an example is encoded in each notation.

Orchestration approaches assume a centralized coordination mechanism which manages the activities that are involved. This means that the activities themselves do not need to be aware that they are part of a composition of activities, and as a consequence, processes do not need to interact with each other [CP11].
The orchestration languages are the following:

- *Sagas*
- *Compensating CSP (cCSP)*
- *StAC*
- *Transaction Calculus*
- *Automata*
- *BPEL*

Choreography approaches do not assume a centralized coordination mechanism, as opposed to orchestration approaches. In this case, activities collaborate together towards forming a higher-level activity through direct communication, meaning that each activity is aware of other collaborators and knows when to start processing, according to some communication sequence. [CP11]
The choreography approaches are the following:

- **πt Calculus**
- **SOCK**
- **Webπ**
- **dcπ**
- **COWS**
- **Committed Join**

3.1.1 Reasoning for selecting the best candidate

The best candidate of the twelve formal languages that I have found was Webπ [MMLI06], for the following reasons:

- This language uses the concept of transactions and compensating transactions by means of a workunit. A workunit $\langle\!|P; Q|\!\rangle_x$ behaves as the body P until an abort signal $\overline{x}$ is signaled and then behaves as the event handler (compensation) Q. This type of behavior is desirable because it directly mimics the behavior of compensating transactions without any other supplementary constructs.
- Webπ implements the concept of protected blocks by using an anonymous workunit $\langle\!|Q; \mathbf{0}|\!\rangle$. This workunit is closed, as it does not have a name, and therefore it cannot be aborted any more. This is also a desirable feature, since Compensable Processes also have the notions of protected blocks which cannot be aborted.
- Both languages, Webπ and Compensable Processes, have similar rules when it comes to LTS. The coincidences and the similarities made the translation somehow easier than if we were to take into consideration another language than Webπ. This will be further illustrated in section §6.1.

More detailed explanations about Webπ can be found in the next section.

3.2 The selected language: Webπ

As mentioned before, the term *orchestration* has been used to address composition and coordination of web services. In order to describe business processes using this approach, several languages have been suggested, and most of them make use of LTS and compensations when dealing with error handling.
Web services are a set of technologies that support the building of integrated and collaborative applications, regardless of the platform that they are built on or the programming languages used to develop them. They provide a platform for the development of applications by using the Internet infrastructure. Web services make their functionalities available over the network, and they can be exploited by other services. A service can itself make use other services and they are all based on the same model. Therefore, composite business processes can be seen as web services, enabling them to be joined in order to form higher-level processes. Thus, web services provide means of building complex systems out of simpler ones.
The orchestration of business processes should follow the satisfy the four primary rules known as ACID[2]:

- **Atomicity** In a transaction involving two or more actions, either all of them are committed or none are.
- **Consistency** A transaction can either lead to another valid state of the data, or, in case of a failure, can return the data to its original state from before the transaction was started.
- **Integrity** A transaction which is taking place and is not yet committed is isolated during its execution

[2] `http://searchsqlserver.techtarget.com/definition/ACID`

- **Durability** Committed changes are saved by the system such that, in the event of a failure and system restart, the data is available in its correct state.

In order to deal with these requirements, Webπ has been introduced as an extension of π-Calculus, which extends the basic calculus with transactional facilities.

3.2.1 The syntax of Webπ

$P ::=$

- (**nil**) $\mathbf{0}$
- (**output**) $\overline{x}\tilde{u}$
- (**guarded choice**) $\sum_{i \in I} x_i(\tilde{u_i}).P_i$
- (**restriction**) $(x)P$
- (**parallel composition**) $P|P$
- (**guarded replication**) $!x(\tilde{u}).P$
- (**workunit**) $\langle\!|P;Q|\!\rangle_x$

A process can be the inert process; $\overline{x}\tilde{u}$ denotes an output of a tuple of names u on a channel x; an input-guarded choice such that upon an input x_i the process evolves into P_i; a process $P$with a restriction on a name x ; a parallel composition; an input guarded replication spawning a process P upon an input on x; a workunit which behaves as P until an abort is signaled on $\overline{x}$, after which it behaves as Q [CP11].

3.2.2 The reduction semantics of Webπ

In order to gain a better understanding of the semantics of Webπ, the following rules had to be taken into consideration. The semantics make use of structural congruence, a relation which equates all the processes that need not be distinguished, and is intended to express basic facts about the operators, such as commutativity or parallel composition [MMLI06].

1. Scope laws

$$\begin{array}{lll} (u)\mathbf{0} & \equiv \mathbf{0}, & \\ (u)(v)P & \equiv (v)(u)P & \\ P(u)Q & \equiv (u)(P|Q), & if u \notin fn(P) \\ \langle\!|(z)P;Q|\!\rangle & \equiv (z)\langle\!|P;Q|\!\rangle, & \text{if} z \notin \{x\} \cup fn(Q) \end{array}$$

2.Workunit laws

$$\begin{array}{ll} \langle\!|\mathbf{0};Q|\!\rangle_x & \equiv \mathbf{0} \\ \langle\!|\langle\!|P;Q|\!\rangle_y|R;S|\!\rangle_x & \equiv \langle\!|P;Q|\!\rangle_y|\langle\!|R;S|\!\rangle_x \end{array}$$

3.Floating laws

$$\langle\!|\overline{z}\tilde{u}|P;Q|\!\rangle_x \equiv \overline{z}\tilde{u}|\langle\!|P;Q|\!\rangle_x$$

The scope laws are standard in π-Calculus while novelties regard workunit and floating laws.
The law $\langle\!|\mathbf{0};Q|\!\rangle_x \equiv \mathbf{0}$ defines a committed workunit, with $\mathbf{0}$ as body. Such a unit cannot fail and is equivalent to $\mathbf{0}$. The law $\langle\!|\langle\!|P;Q|\!\rangle_y|R;S|\!\rangle_x \equiv \langle\!|P;Q|\!\rangle_y|\langle\!|R;S|\!\rangle_x$ moves workunits outside parents, thus flattening the nesting. The last law floats messages outside workunit boundaries. According to this law, messages move towards their inputs, meaning that if a process emits a message, the message traverses the surrounding workunit boundaries until it reaches the corresponding input [MMLI06].

(IN)	$a.P \xrightarrow{a} P$	$x(\tilde{u}).P \xrightarrow{x(\tilde{u})} P$
(OUT)	$\overline{a}.P \xrightarrow{\overline{a}} P$	$\overline{x}\tilde{u} \xrightarrow{\overline{x}\tilde{u}} 0$
(REP)	$\dfrac{P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'\|!P}$	$!x(\tilde{u}).P \xrightarrow{x(\tilde{u})} P\|!x(\tilde{u}).P$
(PAR)	$\dfrac{P \xrightarrow{\alpha} P'}{P\|Q \xrightarrow{\alpha} P'\|Q}$	$\dfrac{P \xrightarrow{\alpha} P', bn(\alpha) \cap fn(Q) = 0}{!P \xrightarrow{\alpha} P'\|!P}$
	(L-RECOVER-IN) $\dfrac{P \xrightarrow{\tilde{t}} P', noComp(P)}{t[P,Q] \xrightarrow{\tau} extr(P')\|\langle Q\rangle}$	(SELF) $\dfrac{P \xrightarrow{\overline{x}} P', inp(P)}{\langle\!\langle P;Q\rangle\!\rangle_x \xrightarrow{\tau} \langle\!\langle xtr(P')\|Q;0\rangle\!\rangle_x}$
	(L-SCOPE-OUT) $\dfrac{P \xrightarrow{\alpha} P', \alpha \neq \lambda x.Q, noComp(P)}{t[P,Q] \xrightarrow{\alpha} t[P',Q]}$	(WUNIT) $\dfrac{P \xrightarrow{\alpha} P', bn(\alpha) \cap (fn(Q) \cup ux) = \emptyset}{\langle\!\langle P;Q\rangle\!\rangle_x \xrightarrow{\alpha} \langle\!\langle P';Q\rangle\!\rangle_x}$
	(L-RECOVER-OUT) $\dfrac{noComp(P)}{t[P,Q] \xrightarrow{t} extr(P)\|\langle Q\rangle}$	(ABORT) $\dfrac{int(P)}{\langle\!\langle P;Q\rangle\!\rangle_x \xrightarrow{x()} \langle\!\langle xtr(P)\|Q;0\rangle\!\rangle}$

Table 1: Comparing the LTSs of Compensable Processes (left) and Webπ (right).

3.2.3 The `xtr()` function

Webπ uses an auxiliary function called extraction function that extracts messages and work units out of the process. It is defined inductively in the following way:

$$
\begin{array}{ll}
\mathtt{xtr}(\mathbf{0}) & = \mathbf{0} \\
\mathtt{xtr}(\overline{x}\tilde{u}) & = \overline{x}\tilde{u} \\
\mathtt{xtr}(\sum_{i\in I} x_i(\tilde{u_i}).P_i) & = \mathbf{0} \\
\mathtt{xtr}((x)P) & = (x)\mathtt{xtr}(P) \\
\mathtt{xtr}(P|Q) & = \mathtt{xtr}(P)\mathtt{xtr}(Q) \\
\mathtt{xtr}(!x(\tilde{u}).P) & = \mathbf{0} \\
\mathtt{xtr}(\langle\!\langle P;Q\rangle\!\rangle_x) & = \langle\!\langle P;Q\rangle\!\rangle_x
\end{array}
$$

3.3 The LTS of Webπ and CP

Because both languages have similar reduction semantics, I have decided to create a parallel table to better show their correspondences. The following table illustrates the LTS of Compensable Processes on the left, and the LTS of Webπ on the right. To note that these are just some of the rules, given as example to illustrate the similarities. In the LTS of Webπ and CP, there are some notations that require further explanation.

1. `noComp(P)` - this predicate holds for Compensable Processes iff process P does not have compensation update which waits for execution. This means that a compensation update has priority over other transitions; that is, if process P in transaction $t[P,Q]$ has a compensation update at the top level then it will be performed before any change of the current state.

2. `inp(P)` verifies whether a process contains an input that is not inside a workunit, which is used to find out whether a unit is still active.
   For example, the predicates $\texttt{inp}(\sum_{i\in I} x_i(\tilde{u_i}).P_i)$ or $\texttt{inp}(!x(\tilde{u}).P)$ hold.

By looking at the LTS of both languages, we can see that some of the rules are similar for both languages. For example, the `IN` rule has a similar behavior in both cases. A process preceded by an action, by taking input will execute the action and result in the standalone process. The rule `REP` is similar to `IN`. A replicated process, by receiving a signal, will extract one of the processes and the rest will continue working. The rule `PAR` states that if a process P is performed in parallel with another process Q, and then P evolves, the new state will keep the process Q working in parallel with the new process P. This rule holds for both languages as well. According to the rules `L-SCOPE-OUT` and `WUNIT`, if a process P evolves into P', this can happen inside a transaction and also inside a workunit. The last rule concerns abortion signals. When a transaction aborts, we preserve the messages inside it and protect the compensation. This is similar for workunits, since the messages are also preserved as well as the compensation, and the workunit may not be aborted any further (therefore, we can say that it is protected).

3.3.1 Example:

In order to illustrate how a work unit can behave in a given situation, let us take the following example:

$$\langle\!|P;Q|\!\rangle_x \xrightarrow{x()} \langle\!|\texttt{xtr}(P)|Q;\mathbf{0}|\!\rangle$$

When a workunit fails, the messages are preserved and also the other workunits inside of it, and this is expressed through the function `xtr`. The compensatory behavior is installed and the workunit becomes anonymous - meaning that it cannot be aborted any more. We can think of anonymous workunits as protected blocks, due to the fact that they cannot be further changed.

4 A new encoding

From the section before, we have seen that there exist similarities between the LTS of the two languages. Therefore, we can proceed further with the design of the new translation. But before doing this, there were a few other aspects that had to be analyzed, since the two languages still had some differences and in order to come up with a proper translation, every aspect had to be treated.

4.0.1 The $\texttt{extr}_P$ and `xtr` functions

Both languages have functions that are concerned with the extraction of protected blocks/processes and messages and work units respectively.
But Compensable Processes have three types of extraction semantics: *discarding, preserving* and *aborting*, defined above, while Webπ only has one. Therefore, in order to find a correct translation between the two, the best fit between the three extraction semantics of compensable processes had to be found for the one in Webπ.
By analyzing the four discarding semantics, I have come to the conclusion that the `xtr` function has the best fit with the preserving semantics of CP. While `xtr` is concerned with preserving the messages and work units out of a process, $\texttt{xtr}_P$ preserves protected blocks and top level transactions.
The discarding semantics of CP would have resulted in a loss of information due to the fact that they only

preserve protected blocks, and the aborting semantics preserve too much information because they keep the nested transactions as well.

4.0.2 The $nw(P)$ function

In order to correctly preserve the work units inside a process P, we had to count them. Therefore, we came up with the function $nw(P)$ which counts the number of protected units in P, and can be defined inductively in the following way:

$$\begin{aligned}
nw(\mathbf{0}) &= \mathbf{0} \\
nw(\overline{x}\tilde{u}) &= \mathbf{0} \\
nw(x(\tilde{u}).P) &= nw(P) \\
nw((x)P) &= nw(P) \\
nw(P|Q) &= nw(P) + nw(Q) \\
nw(!x(\tilde{u}).P) &= nw(P) \\
nw(\langle\!|P;0|\!\rangle) &= 1 \\
nw(\langle\!|P;Q|\!\rangle_x) &= 1 + nw(P)
\end{aligned}$$

For example, if we had the following workunit: $P = \langle\!|\langle\!|P';0|\!\rangle;Q|\!\rangle_x$,
then $nw(P) = nw(\langle\!|\langle\!|P';0|\!\rangle;Q|\!\rangle_x) = 1 + nw(\langle\!|P';0|\!\rangle) = 2$

4.0.3 The list of outputs L

We have decided to maintain a set $L(P)$ which contains the list of outputs in a process P. This decision is based on the fact that the extraction function in Webπ preserves messages, so in $L(P)$ we will preserve the outputs.

$$\begin{aligned}
L(\mathbf{0}) &= \mathbf{0} \\
L(\overline{x}\tilde{u}) &= \{x\} \\
L(x(\tilde{u}).P) &= L(P) \\
L((x)P) &= L(P) \\
L(P|Q) &= L(P) \cup L(Q) \\
L(!x(\tilde{u}).P) &= L(P) \\
L(\langle\!|P;Q|\!\rangle_x) &= L(P)
\end{aligned}$$

4.1 The auxiliary encoding

In order to define our translation, denoted $[\![Q]\!]_\rho$ where ρ is a path, we needed to define an auxiliary encoding, which can be found below. Both the auxiliary encoding and the encoding itself are mutually defined, being used in each other's definitions. In our case, since the extraction function in Webπ preserves messages and work units out of the processes, our auxiliary encoding, denoted $\parallel Q \parallel_{x,\rho}^{L,n}$ has two parameters. L denotes the set of messages (outputs), and n the number of protected blocks in P.
We can introduce the notion of $path$,a sequence that which will describe the location of a process. For example, process $P = \langle\!|R;Q|\!\rangle_x$, then process R is situated at location x. If we take into consideration another example, where process $P_2 = \langle\!|\langle\!|R;Q_1|\!\rangle_y;Q|\!\rangle_x$, then process R is situated at location x, y.

Definition: *Let Q be a compensating process, ρ be a path and x the name of the workunit where Q*

lives. The process $\| Q \|^{L,n}_{x,\rho}$ *is defined based on the encoding* $[\![Q]\!]$ *and is defined as:*

$$\| Q \|^{\varnothing,0}_{x,\rho} = l_x.\overline{m_x}.a.\beta_\rho[n[[\![Q]\!]_\rho]]|m_x.k_x.x\{\dagger\}.o\{\dagger\}$$

$$\| Q \|^{\{z\},\mathbf{0}}_{x,\rho} = l_x.o^z_{x,\rho}\{(X).z\{a.o_z[X]|\overline{m_x}.\beta_\rho[n[[\![Q]\!]_\rho]]|o^x_\rho\{\dagger\}]\}\}.(z[0]|m_x.\overline{k_x}.x\{\dagger\})$$

$$\| Q \|^{\varnothing,1}_{x,\rho} = l_x.\beta_{x,\rho}\{(Y).z\{a.\beta_u[Y]|\overline{m_x}.\beta_\rho[n[[\![Q]\!]_\rho]]|o_x[\{\dagger\}]\}\}.(z[0]|m_x.\overline{k_x}.x\{\dagger\})$$

$$\begin{aligned}\| Q \|^{\{z\},1}_{x,\rho} = {} & l_x.o^z_{x,\rho}\{(X).\beta_{x,\rho}\{(Y).w\{o^z_\rho[X]|a(\beta_\rho[Y])|\overline{m_x}.\beta_\rho[n[[\![Q]\!]_\rho]]\}\}\}.\\ & (w[0]|m_x.(\overline{k}_x.x\{\dagger\}|o^x_\rho\{\dagger\})\end{aligned}$$

$$\begin{aligned}\| Q \|^{\{z,t..u\},n}_{x,\rho} = {} & l_x.o^z_{x,\rho}\{(X_1).o^t_{x,\rho}\{(X_2)....o^u_{x,\rho}\{(X_u)\}\}.\\ & \beta_{x,\rho}\{(Y_1,Y_2...Y_n).w\{o^z_\rho[X_1]|o^t_\rho[X_2]|...|o^u_\rho[X_u]|\\ & a.(\beta_\rho[Y_1]|\beta_\rho[Y]_2|...|\beta_\rho[Y_n])|\overline{m_x}.\beta_\rho[n[[\![Q]\!]_\rho]]\}\}\}.\\ & (w[0]|m_x.(\overline{k}_x.x\{\dagger\}|o^x_\rho\{\dagger\})\end{aligned}$$

The first encoding $\| Q \|^{0,0}_{x,\rho}$ represents the auxiliary encoding of a process that has no outputs and workunits. The next two represent encodings when the process has exactly one output and one workunit respectively. The last encoding is concerned with the situation when the process has multiple outputs and multiple workunits.
These situations had to be treated separately because it is important to know the number of messages and units, so that they can be properly preserved and to prevent losing information.

4.1.1 Abbreviations

Before providing the encoding itself, the following abbreviations for update prefixes need to be mentioned:

- $t\{\dagger\}$ for the update prefix $t\{(Y).\mathbf{0}\}$ which "kills" location t together with all the processes in that location.
- $t\{P\}$ for the update prefix $t\{(Y).P\}$ that replaces the current behavior at t with P.
- $t\{id\}$ for the update prefix $t\{(X).X\}$ which deletes the location name t;
- $t\{(X_1, X_2, ...X_n).R\}$ for the sequential composition of updates $t\{(X_1).t\{(X_2).....t\{(X_n).R\}\}\}$

4.2 The translation

In order to provide the complete translation, we need to take into consideration all the constructs present in Webπ.
Definition: *Let P and Q be web processes and let* ρ *be a path. The encoding* $[\![.]\!]_\rho$ *of Web*π *processes into Adaptable Processes is defined as:*

- $[\![\mathbf{0}]\!]_\rho = \mathbf{0}$
- $[\![\overline{x}\tilde{u}]\!]_\rho = o^x_\rho[\overline{x}(\tilde{u})]$
- $[\![\overline{x_1}(\tilde{u_1}).P_1 + \overline{x_2}(\tilde{u_2}).P_2]\!]_\rho = [\![\overline{x_1}(\tilde{u_1}).P_1]\!]_\rho + [\![\overline{x_2}(\tilde{u_2}).P_2]\!]_\rho$
- $[\![(x)P]\!]_\rho = (x)[\![P]\!]_\rho$
- $[\![P|Q]\!]_\rho = [\![P]\!]_\rho | \ [\![Q]\!]_\rho$

- $[\![!x(u).P]\!]_\rho = !x(u)[\![.P]\!]_\rho$
- $[\![(\!|P; \mathbf{0}|\!)]\!]_\rho = \beta_\rho[n[[\![P]\!]]]$, where n has not been used anywhere else
- $[\![(\!|P; Q|\!)_x]\!]_\rho = \beta_\rho[x[[\![P]\!]_\rho]| \parallel Q \parallel_{x,\rho}^{L,nw(P)} |x().\overline{l_x}.k_x.\overline{j}.0]|j.\beta_x\{id\}.\overline{a}.\mathbf{0}$

The encoding of the null process is going to be, intuitively, the null process.
The rule $[\![\overline{x}\tilde{u}]\!]_\rho = o_\rho^x[\overline{x}(\tilde{u})]$ says that the encoding of a message unit is going to be a location o, identified by the name of the message as well as the path where it was found, which will contain the respective message. This information is important to be saved, since the messages need to be preserved and we need to know where to find them after the encoding.
The encoding of a guarded choice between two processes is the same as the choice between the encoding of the two separate processes. This is due to the fact that the two processes take place concurrently and according to the principle of compositionality, which states that the translation of a complex expression is determined by the translation of the constituent expressions.
The next encoding, $[\![(x)P]\!]_\rho = (x)[\![P]\!]_\rho$ means that by encoding a restriction on a process, the input can be taken outside of the encoding and then the process itself is encoded, since the input will not influence the encoding of the process.
The encoding of the parallel composition of two processes is the same as the parallel composition of the two processes encoded separately, also following the principle of compositionality.
Similar to the encoding of the guarded choice, the guarded replication can be encoded by taking the messages outside of the encoding and encoding the process only.
The encoding of an anonymous workunit is a location β, which will contain another location n, and here is where the encoding of the process P will reside. An anonymous work unit has no name and no output messages, therefore there is no need for this information to be preserved. The location n should not have been used anywhere else, as we have decided to use it solely for the purpose of encoding anonymous units.
The last encoding represents the encoding of a normal work unit. This encoding results in the composition of four processes. The leftmost process encodes the default activity P preserving the nested structure. The two rightmost components will execute the auxiliary encoding component, which will find all the top-level transactions and messages in the compensatory behavior and try to preserve them, by moving them to a specific location. When coming up with the translation for the work unit, we have tried to keep a similar structure of the components as the one used in the translation between Compensable Processes and Adaptable Processes, which was previously done in the paper [DPP15]. The reasoning behind this is two-folded. Firstly, by relying on previous work we can better ensure the correctness of the new encoding, and secondly, having similar structures between encodings would be beneficial for future work, if we were looking to find a translation between other languages.

5 Example

In order to illustrate the translation and how it occurs, we can look at an example and analyze it.
We will try to prove that our example also satisfies the property of operational correspondence.

Example 5.1. *Let P_0 be a web process, such that $P_0 = \langle\!|\overline{z}(w)|Q; R|\!\rangle_x|\overline{x}(v)$, with nw(Q) = 0.*

Then $P_0 \xrightarrow{\tau} \langle\!|R; 0|\!\rangle|\overline{z}(w)$. We will expand based on the translation. Recall that we omit $\mathbf{0}$ wherever possible.

$$
\begin{aligned}
P_0 \quad &= \langle\!|\overline{z}(w)|Q; R|\!\rangle_x|\overline{x}(v) \\[2ex]
[\![P_0]\!]_\epsilon \quad &= [\![\langle\!|\overline{z}(w)|Q; R|\!\rangle_x|\overline{x}(v)]\!]_\epsilon \\
&= [\![\langle\!|\overline{z}(w)|Q; R|\!\rangle_x]\!]_\epsilon|[\![\overline{x}(v)]\!]_\epsilon \\
&= \beta_\epsilon\Big[x\big[[\![\overline{z}(w)|Q]\!]_\epsilon\big]| \ \parallel R \parallel_{x,\epsilon}^{\{z\},0} \ | \ x().\overline{l_x}.k_x.\overline{j}\Big]|j.\beta_x\{id\}.\overline{a} \mid o_\epsilon^x[\overline{x}(v)] \\[2ex]
&= \beta_\epsilon\Big[x\big[o_\epsilon^z[\overline{z}(w)]|[\![Q]\!]_\epsilon\big]| \\
&\quad l_x.o_\epsilon^z\{(X).z\{a.o_\epsilon^z[X] \mid \overline{m_x}.\beta_{x,\epsilon}\big[n[[\![R]\!]_\epsilon] \mid o_\epsilon^x\{\dagger\}\big]\}\}.(z[0] \mid m_x.\overline{k_x}.x\{\dagger\})| \\
&\quad x().\overline{l_x}.k_x.\overline{j}\Big] \mid j.\beta_x\{id\}.\overline{a} \mid o_\epsilon^x[\overline{x}(v)] \\[2ex]
\rightarrow^* \quad &\beta_\epsilon\Big[x\big[[\![Q]\!]_\epsilon\big]| \ k_x.\overline{j} \mid a.o_\epsilon^z[\overline{z}(w)] \mid \overline{m_x}.\beta_{x,\epsilon}\big[n[[\![R]\!]_\epsilon] \mid o_\epsilon^x\{\dagger\}\big]|m_x.\overline{k_x}.x\{\dagger\} \ \Big] \mid \\
&j.\beta_x\{id\}.\overline{a} \mid o_\epsilon^x[\mathbf{0}] \\[2ex]
\rightarrow^* \quad &\beta_\epsilon\Big[x\big[[\![Q]\!]_\epsilon\big]| \ a.o_\epsilon^z[\overline{z}(w)] \mid \beta_{x,\epsilon}\big[n[[\![R]\!]_\epsilon] \mid o_\epsilon^x\{\dagger\}\big]| \ x\{\dagger\} \ \Big] \mid \beta_x\{id\}.\overline{a} \mid o_\epsilon^x[\mathbf{0}] \\[2ex]
\rightarrow^* \quad &x\big[[\![Q]\!]_\epsilon\big]| \ o_\epsilon^z[\overline{z}(w)] \mid \beta_{x,\epsilon}\big[n[[\![R]\!]_\epsilon] \mid o_\epsilon^x\{\dagger\}\big]| \ x\{\dagger\} \mid o_\epsilon^x[\mathbf{0}] \\[2ex]
\rightarrow \quad &o_\epsilon^z[\overline{z}(w)] \mid \beta_{x,\epsilon}\big[n[[\![R]\!]_\epsilon] \mid o_\epsilon^x\{\dagger\}\big]| \ o_\epsilon^x[\mathbf{0}] \\[2ex]
\rightarrow \quad &o_\epsilon^z[\overline{z}(w)] \mid \beta_{x,\epsilon}\big[n[[\![R]\!]_\epsilon]\big] \\
&\equiv [\![\overline{z}(w)]\!]_\epsilon \mid [\![\langle\!|R; 0|\!\rangle]\!]_\epsilon \\
&\equiv [\![\overline{z}(w) \mid \langle\!|R; 0|\!\rangle]\!]_\epsilon
\end{aligned}
$$

From the literature, we know that the following statement of operational correspondence holds [DPP15].

Let ρ be an arbitrary path. Then:

$$\textit{If } P \xrightarrow{\tau} P' \textit{ then } [\![P]\!]_\rho \rightarrow^* [\![P']\!]_\rho$$

To note is that $\rightarrow^*$ symbolizes that a number of steps (1 or more) need to be performed in the transition. Through our example, we have also shown that if the property proven in the other paper also holds for my encoding. This example is just one of four cases that need to be treated, but the complete, formal proof for the correctness of the encoding is subject for future work. Thus, in our example, we have shown that if $P_0 \xrightarrow{\tau} \langle\!| R; 0 |\!\rangle | \overline{z}(w)$ then $[\![P_0]\!]_\epsilon \rightarrow^* [\![\langle\!| R; 0 |\!\rangle | \overline{z}(w)]\!]_\epsilon$, therefore proving the property of operational correspondence of the translation.

6 Conclusion

My research was based on the comparison between two related but yet fundamentally different process models: the calculus of Adaptable Processes and Webπ. I have provided the encodings of processes into Adaptable Processes. The encodings are not only non-trivial applications of web processes as present in Webπ, but they also help gain a better understanding of the semantics of Adaptable processes. Compositionality and more importantly, operational correspondence have been taken into consideration when creating the encodings.

My research has opened up several possibilities for future work. Having addressed the translation only from one language to the other, an interesting topic would be the reverse direction, and whether that would be possible or not.

References

[AILS07] Luca Aceto, Anna Ingólfsdóttir, Kim Guldstrand Larsen, and Jiri Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press New York, NY, 2007.

[CP11] Christian Colombo and Gordon J. Pace. A compensating transaction example in twelve notations. Technical report, University of Malta, 2011.

[DPP15] Jovana Dedeic, Jovanka Pantovic, and Jorge A. Pérez. On compensation primitives as adaptable processes. In *Proc. EXPRESS/SOS Madrid, Spain*, pages 16–30, 2015.

[MMLI06] Mazzara, Manuel, Lanese, and Ivan. *Towards a Unifying Theory for Web Services Composition*, pages 257–272. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.