UNIVERSITY OF

Southampton

University of Southampton Research Repository
ePrints Soton

http://eprints.soton.ac.uk

UNIVERSITY OF SOUTHAMPTON

FACULTY OF ENGINEERING, SCIENCE AND MATHEMATICS

School of Electronics and Computer Science

Extending and Relating Semantic Models of Compensating CSP

by

Shamim H Ripon

Thesis for the degree of Doctor of Philosophy

August 2008

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING, SCIENCE AND MATHEMATICS
SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

EXTENDING AND RELATING SEMANTIC MODELS OF COMPENSATING CSP

by Shamim H Ripon

Business transactions involve multiple partners coordinating and interacting with each other. These transactions have hierarchies of activities which need to be orchestrated. Usual database approaches (e.g.,checkpoint, rollback) are not applicable to handle faults in a long running transaction due to interaction with multiple partners. The compensation mechanism [54] handles faults that can arise in a long running transaction. Based on the framework of Hoare's CSP process algebra [59], Butler *et al* [27], introduced Compensating CSP (cCSP), a language to model long-running transactions. The language introduces a method to declare a transaction as a process and it has constructs for orchestration of compensation. Butler *et al* also defines a trace semantics for cCSP.

In this thesis, the semantic models of compensating CSP are extended by defining an operational semantics, describing how the state of a program changes during its execution. The semantics is encoded into Prolog to animate the specification. The semantic models are further extended to define the synchronisation of processes. The notion of partial behaviour is defined to model the behaviour of deadlock that arises during process synchronisation. A correspondence relationship is then defined between the semantic models and proved by using structural induction. Proving the correspondence means that any of the presentation can be accepted as a primary definition of the meaning of the language and each definition can be used correctly at different times, and for different purposes.

The semantic models and their relationships are mechanised by using the theorem prover PVS [89]. The semantic models are embedded in PVS by using Shallow embedding [100]. The relationships between semantic models are proved by mutual structural induction. The mechanisation overcomes the problems in hand proofs and improves the scalability of the approach.

Several case studies are carried out to model web services by using cCSP constructs that shows the expressiveness of cCSP. It is shown how cCSP can be used to model the coordination of several web services along with defining synchronisation between and within transaction blocks.

Contents

List of Figures

List of Tables

Acknowledgements

First and foremost, I would like to thank my supervisor, Professor Michael Butler, for his guidance, constant encouragement and keen eye for details helping me to simplify various problems. Many thanks to Neil Evans for assisting at the early stage of theorem proving in PVS. Thanks to The Charles Wallace Bangladesh Trust for supporting me during the writing of this thesis.

Many thanks to the members of DSSE research group for many pleasant and fruitful discussions. Finally, thanks to my family, especially Tuli for her endless support and inspiration. I dedicate this thesis to my heavenly father.

Chapter 1

Introduction

1.1 Introduction and Motivation

Business transactions typically involve coordination and interaction between multiple partners. These transactions involve hierarchies of activities and these activities need to be orchestrated. Business transactions need to deal with faults that can arise in any stage of the transactions. In usual database transactions, a rollback mechanism is used to handle faults in order to provide atomicity to a transaction [54]. However, for transactions that require long periods of time to complete, also called *Long Running Transactions*(LRT), rollback is not always possible. LRTs are usually interactive (communication with several agents). Handling faults where multiple partners are involved are both difficult and critical. Due to their interactive nature, LRTs are not able to be check-pointed, e.g. a sent message cannot be 'unsent'. In such cases, a separate mechanism is required to handle faults. A possible solution of the problem would be that the system designer can provide a mechanism to compensate the actions that can not be undone automatically.

Compensation is defined in [54] as an action taken to recover from error in business transactions or cope with a change of plan. Consider the example: a customer buys some items from an on-line store. The store debits the customer's account for the payment of the items. Later the store realises that one or more items are not available at that time. So, to compensate the customer, the store can credit the already debited amount and at the same time notify the customer their apology, or the store can take alternate actions, such as, arranging items from an alternative source or asking the customer whether they want a later delivery, etc. The scenario shows that the concept of compensation is more general than traditional database rollback. Compensations are very important for handling failures in long running transactions. Compensations are installed for every committed activity in a long-running transaction. The idea of compensation was introduced in [49] to define the concept of *sagas*. A saga partitions

a long running transaction into a sequence of sub-transactions. Each sub-transaction has an associated compensation. If one sub-transaction fails, then compensations of the committed sub-transactions in the sequence are executed in reverse order.

Web services technology provides a platform on which we can develop distributed services. The interoperability among these services is achieved by the standard protocols (WSDL [32], UDDI [88], SOAP [36]) that provide the ways to describe services, to look for particular services and to access services. With the emergence of web services, *business transactions* are conducted using these services [67]. Web services provided by various organisations can be inter-connected to implement business collaborations, leading to composite web services.

Business collaborations require interactions driven by explicit process models. Web services are distributed, independent processes which communicate with each other through the exchange of messages. The coordination between business processes are particularly crucial as it includes the logic that makes a set of different software components become a whole system. Hence it is not surprising that these coordination models and languages have been the subject of thorough formal study, with the goal of precisely describing their semantics, proving their properties and deriving the development of correct and effective implementations.

Process calculi are models or languages for concurrent and distributed interactive systems. It has been advocated in [72, 103] that process algebras provide a complete and satisfactory assistance to the whole process of web services development. Being simple, abstract, and formally defined, process algebras make it easier to formally specify the message exchange between web services and to reason about the specified systems. Transactions and calculi have met in recent years both for formalising protocols as well as adding transaction features to process calculi [13, 15, 16, 20].

Several research issues, both theoretical and practical, are raised by web services. Some of the issues are to specify web services by a formally defined expressive language, to compose them, and to ensure their correctness; formal methods provide an adequate support to address these issues [41]. Recently, many XML-based process modelling languages (also known as *choreography* and *orchestration* [95] languages) such as WSCI [2], BPML [1], BPEL4WS [35], WSFL [66], XLANG [113] have emerged that capture the logic of composite web services. These languages also provide primitives for the definition of business transactions.

Several proposals have been made in recent years to give a formal definition to compensable processes by using process calculi. These proposals can be roughly divided into two categories. In one category, suitable process algebras are designed from scratch in the spirit of orchestration languages, e.g., BPEL4WS. Some of them can be found in [21, 24, 27]. In another category, process calculi like the π-calculus [79, 93] and the join-calculus [46] are extended to describe the interaction patterns of the services where,

each service declares the ways to be engaged in a larger process. Some of them are available in [16, 20, 63, 69]. In this thesis, we are particularly interested in the former category. Developing an algebra requires to develop semantic models for that algebra. This thesis is aimed at contributing to the development of the formal semantics of an algebra in the former category, where an algebra is defined to model business transactions with orchestration of compensations.

A formal semantics offers a complete and rigorous definition of a language. There are three main approaches to defining the semantics of a language, namely, operational, denotational and axiomatic semantics. Operational semantics [56, 96] specifies the way in which programs run by means of abstract execution rules. A denotational semantics [50, 104] description of a programming language defines the meaning of the language in terms of mathematics objects. Axiomatic semantics [43, 57] defines a language by providing assertions and inference rules for reasoning about programs. No single semantic framework is ideal with regard to all pragmatic aspects, and it is also suggested in [82] that two or more complementary semantic descriptions should be provided for a language.

Several proposals have been made in recent years where process algebras are defined to model business transactions. We have already discussed that while defining a language, two or more complementary semantic descriptions should be provided for it in order to establish a formal foundation for the language. As mentioned earlier, this thesis will focus on the formalism of a process algebra which is designed by the sprit of orchestration language. Among the algebras in this category, both StAC [24, 42] and *Sagas* [21] are defined by giving their operational semantics. On the other hand, compensating CSP [27] is defined by giving its denotational (trace) semantics. cCSP has been defined with inspiration from the standard CSP [59] process algebra and transaction processing features to model long running transactions, combined with orchestration of compensations. StAC, which is a predecessor of cCSP, provides a large number of operators and the operational semantics is defined in terms of an intermediate language, called $StAC_i$ [25]. The complex semantic definition of $StAC_i$ prevents the definition of a simple compositional semantics. Comparing to that, the trace semantics of cCSP provides a compositional semantic definition resulting in a neater semantic model than StAC. Unlike StAC, in cCSP the invocation of compensation for a transaction block is automatic in the case of an exception in the block, while compensations are discarded for a successful termination of a block. It is apparent from the discussion that cCSP is a good candidate for our experiment in the modelling of business transactions. Having a trace semantics of the language also gives us a good starting point to understand the algebra as well as to extend it further.

Inspired by the growing interest of transaction processing features, and having the intention to provide process algebraic support to model business transactions, our goal in this thesis is to provide a formal foundation to the modelling of long running transactions,

and we carry on our experiments by using cCSP. Our main objectives in this thesis are:

- Extend the existing semantic models of compensating CSP.

- Define and prove a formal relationship between the semantic models, which will strengthen the formal foundation of the language.

- Contribute to the language design by adopting a systematic approach to having multiple semantic models and mechanising their relationship.

1.2   Main Contributions

Keeping in mind the objectives defined in the previous section, the main contributions of our work lie in the extension of semantic models of cCSP and then establishing a formal relationship between the two defined semantic models. In particular, we define an operational semantics for cCSP and then define and prove a relationship between the denotational and the operational semantics of cCSP. Added to this, we mechanise the semantic models and the proofs of their relationship by using a theorem prover (PVS [89]). The two semantic models and their relationship establish a strong formal foundation for cCSP. In the rest of the section, we detail the main contribution of this thesis.

- We extend the existing semantic model by defining an *operational semantics* of Compensating CSP. The semantics is defined by using labelled transition systems following the approach shown by Plotkin [96]. The operational semantics defines how the states of a program change during execution. We then encode the operational semantic by using the tool XTL [8], which allows the animation of the specification in cCSP.

- After defining the operational semantics, we have two semantic models of cCSP. Having these two semantic models, it is natural to see how these two semantic models are *related to each other*. We define and prove a formal relationship between the denotational and the operational semantics. We adopt a systematic approach where we extract traces from the transition rules defined in the operational semantics, and then prove by structural induction that the derived traces correspond to the originally-defined traces. The important aspect of showing the correspondence between the semantic models is that it gives consistency to the language and shows that both the definitions are significant in their own areas of applications.

- At the initial stage of the cCSP development, we avoided defining the semantics of synchronisation for concurrent processes. Synchronisation is a well understood

and significant feature for distributed processes. We extend the semantic models
further to define the semantics of a synchronisation of processes, where processes
synchronise over a set of synchronising events, and non-synchronising processes in-
terleave with each other. As we mechanise the cCSP semantics in a later chapter,
we do not perform the detailed proofs by hand, instead we only outline the rela-
tionship between the operational and trace semantics for synchronous processes.
We consider the cases where synchronisation leads to deadlock and we show how
to extract partial behaviour from the processes that lead to deadlock.

- The proofs showing the correspondence between the semantic models are carried
  out completely by hand, which is an error prone and tedious task, and it is diffi-
  cult to handle large number of steps. In order to overcome these problems in hand
  proofs and at the same time to find a feasible mechanisation for the semantic mod-
  els and their properties, we mechanise the semantic models and the relationship
  showing their correspondence. First, both semantic models are embedded in the
  theorem prover PVS [89] and we use a *shallow embedding* [100] for this purpose.
  We then define the theorems and supporting lemmas that prove the relationship
  between the models. The proofs are carried out by using mutual structural in-
  duction. At the initial stage of mechanical proofs we followed a similar level of
  steps as our hand proofs. In later stages, the proofs were carried out without
  any support from the hand proofs. In particular, the semantic correspondence
  for synchronisation of concurrent processes, which is extended in a later stage of
  cCSP development, was carried out without any corresponding hand proofs. For
  synchronous processes, we considered all the cases where processes synchronise,
  interleave and even the cases when processes deadlock.

- We perform case studies by modelling web services in order to investigate the
  expressiveness of cCSP constructs, especially the orchestration of compensation
  and synchronisation between synchronising processes. We model two separate ver-
  sions of the same web service, which allows us to experiment with synchronisation
  between and within transactions blocks as well as handling of compensations.

1.3   Thesis Structure

In the rest of the thesis, we survey the existing literature related to our work and describe
in detail how we have made the contribution outlined in the previous section. The is
achieved through the course of the remaining chapters which are structured as follows:

- Chapter 2 introduces the technical background and concepts that are essential for
  the notations and formal contents of this thesis. In doing so, we briefly describe
  the basic process algebraic concepts and notation on which our algebraic approach

is built. The chapter describes the notion of long running transactions in web services and identifies the relevant existing literature on applying formal methods to describe them. Considering our objectives mentioned earlier, we identify the limitations of existing works and use these to motivate the work described in the subsequent chapters.

- Chapter 3 defines the operational semantics for compensating CSP. The operational semantics is defined by using labelled transition systems. Semantics are defined for both standard and compensable processes (processes with attached compensations). The operational semantics is defined by small step semantics so that it is possible to observe each step of the execution and how each process terminates. The semantics are encoded in Prolog in order to animate the specification in XTL. A brief outline of the Prolog encoding is also described in the chapter.

- Chapter 4 shows the relationship between the operational and the trace semantics. The relationship is derived for both standard and compensable processes. After stating the theorems that define the correspondence between the semantic models, the supporting lemmas are derived from the definition of traces and transition rules of each process term. Induction is applied over traces to prove the correspondence for each process term. The semantic correspondence is then proved by structural induction over process terms.

- Chapter 5 further extends the semantic models by defining the operational rules for synchronisation of concurrent processes. The parallel operator was defined in [27] as interleaving where processes interleave with each other and synchronise only on termination. The semantics is extended to define the synchronisation of processes on observable events. Deadlock arises when the synchronising events do not synchronise with each other. The semantic correspondence of the synchronous processes is also proved by following similar approaches to those shown in the previous chapter. A special terminal event is introduced to represent the partial behaviour of processes that do not synchronise over synchronising events.

- Chapter 6 describes the mechanisation of the semantic models and their relationship for both standard and compensable processes, by using the theorem prover PVS. First, both semantic models are mechanised in PVS using a shallow embedding. The theorems and the supporting lemmas are then defined following the same approach as in the hand proofs. The proofs are carried out by following similar steps as in the hand proofs, while in a later stage the proofs are carried out without any support from the hand proofs. The chapter shows how the mechanisation helps to identify some theorems that are ignored in the hand proofs, and how the semantic models are improved during the mechanisation.

- Chapter 7 describes the case studies performed to model two separate versions of a Car Broker web service by using cCSP constructs. The Car Broker web service

has two associated web services. The chapter also shows the modelling of these two web services and how all these web services are composed together to model the whole system. The case study particularly highlights the interactions between process synchronisation and compensations.

- Finally, Chapter 8 summarises the conclusions drawn throughout the thesis and in particular the contributions and limitations of our works. We also describe and motivate our plans for future work.

Chapter 2

Background

In this chapter we present a review of the literature relevant to understanding and evaluating our work. The chapter is divided into three main parts. First, Section 2.1 puts our motivation in context by giving an overview of relevant process algebras. Then, Section 2.2 gives a brief overview of Web services covering to the use of process algebra in formalising business transactions in web services. Next, we outline the concept of transaction processing using process algebra, and then describe the idea of modelling long running transactions and show some of the recent work on applying process algebra in modelling long running transactions.

2.1 Process Algebra

The term 'Process Algebra' can be used with different meanings. Before describing process algebra let us start with the term 'process'. A process can be referred to as the behaviour of a system. The behaviour of the system can be in particular the execution of a software system, the action of a machine or even the actions of a human being. The word 'algebra' refers to an algebraic/axiomatic approach in talking about the behaviour. We can say that process algebra refers to mathematical formulation and reasoning rules for defining processes, made up of events where we focus on communication events which occur between processes. Details of process algebra and it history can be found in [9, 14].

Several notions and formalisms have been defined for process algebra. Some of the basic formalisms are CCS [77], CSP [59], ACP [10]. Some of the extensions are the π-calculus [93], Timed CSP [106]. Although all these process algebras are syntactically different, they share a set of basic constructs: actions, sequential and parallel composition, synchronised actions, choice and some other basic activities. In this description, our main focus will be on CSP, and this will be followed by a glimpse of some of the other widely used process algebras.

CSP

CSP, developed by C. A. R. Hoare [59], is a language for describing concurrent systems, whose component processes interact with each other by communication. In CSP, systems are modelled as processes. Each process can communicate with other processes and the environment through events. The set of events associated with each process is called the alphabet of that process. The behaviour of a process is described by an algebraic expression involving CSP operators.

The CSP language includes several basic processes. The simplest CSP processes are STOP and SKIP. STOP does nothing and never communicates, and SKIP is successful termination of a process. The prefix operator ($\longrightarrow$) is used for sequencing of events, and the expression $a \rightarrow P$ describes a process that engages in the event a and then behaves as P. External choice is described by the choice operator $\Box$: $P \Box Q$ describes a process that can behave either as P or as Q, and the choice is resolved by environment. The internal choice operator is designated by $\sqcap$ and the expression $P \sqcap Q$ describes a process that behaves like either P or Q where the choice is resolved internally, not by the environment.

The parallel composition of two processes is represented as $P \parallel Q$. P interacts with Q by synchronisation over shared events that are common to both P and Q. However, this can lead to deadlock when no process can make any progress as they do not have any common next event. Interleaving is written as $P \mathbin{|||} Q$ where P and Q are executed independently of each other. Sequential composition is represented as $P \mathbin{;} Q$ where Q will execute only after P is completed.

Processes can be defined recursively. The recursive expression $\mu P.F(P)$ behaves as $F(P)$ where $F(P)$ is a guarded expression containing P. The hiding operator is used to hide events from the environment. The expression $P \backslash X$ behaves like P and the events in X are hidden.

The CSP trace model [59] represents the behaviour of processes as sequences of observable events. The internal and external choice cannot be distinguished by their traces. They can be distinguished by their *refusals*. A refusal is a set of events that a process fails to accept. A finer distinction between the processes than the one made by refusals can be made by using *failures*. A refusal tells what a process can refuse after an empty trace, whereas a failure tells what a process can refuse after any of its trace. A failure of a process is a pair (s, X), where $s \in traces(P)$ and $X \in refusals(P/s)$ (refusals of P after trace s). $failure(P)$ is the set of failures of P. The failure model does not allow us to detect *livelock* (i.e., an infinite sequence of internal actions). The failure-divergences model can detect it by adding the concept of *divergence*. The divergences of a process are the set of traces after which the process may livelock. Refinement relations can be defined for systems described in CSP for traces, failures and failure-divergences.

The ProBE [45] tool can be used to animate arbitrary CSP process descriptions,

and the FDR [44] tool supports automated refinement checking of CSP processes.

CCS

The Calculus of Communicating Systems (CCS), the process algebraic language, was developed by Robin Milner [77]. Like CSP, it can describe the structures and behaviours of concurrent systems. In CCS a system is viewed as a collection of interacting processes where the behaviour of each process is defined by using an expression in CCS. The composition of all individual processes defines the whole system. Although CCS has similar constructs to CSP, there are significant differences in the interpretation of some operators. CCS has some basic operators for composition of processes, which are: prefix to express sequential actions, summation to express both deterministic and non-deterministic selections, composition for concurrency, renaming for syntactic variants and restriction for internal actions.

Communication is the exchange of a message between processes and in CCS the only interaction among the processes is communication and communication is atomic and synchronous. The CCS semantics is defined by a set of transition rules. Whenever a process engages in an event it makes a transition. Each process operator has some associated transition rules that define the meaning of the operator. The notion 'Bisimulation' defines the behavioural equivalence over processes by using their transition sequences.

As mentioned in [58], the divergence between CSP and CSS is on purpose as they were developed for their own purposes. A detailed description of their differences is beyond the scope of this thesis. Detailed comparisons of these two calculi can be found in [119].

π-calculus

The π-calculus is a mathematical model of processes whose interconnections change as they interact. Communication links are transferred between two processes and the recipient can use the link for further interaction with other parties. This kind of communication provides the facility to model systems whose accessible resources vary over time. The π-calculus was first presented in [79] to model concurrent communicating systems and it is an extension of the CCS process algebra. π-calculus is the foundation of two of the main Process Markup Languages: BPML from the BPMI consortium and XLANG [113] (now BPEL4WS)[35] from Microsoft.

2.2 Web Services

The term 'web services' describes a standardised way of integrating web-based applications using open standards (e.g., XML, SOAP and UDDI) over an internet protocol backbone. Used primarily as a means for businesses to communicate with each other

and with clients, web services allow organisations to communicate data without intimate knowledge of each others IT system behind the firewall. Instead of providing GUIs to users, web services share business logic, data and processes through a programmable interface across the network.

In order to create a cross-organisational component in web services, flexible methods are needed to handle web service interfaces. To describe the composition of web services as well as the process flow, two terms are widely used: *Orchestration* and *Choreography*. How executable business processes interact with each other is described by orchestration. It also describes the business logic, execution order and all the interactions are at the message level. These interactions can span applications and organisations and result in a long-running, transactional and multi-step process model.

In terms of web services where multiple parties are involved, orchestration represents control from one party's perspective. Choreography, on the other hand, tracks massages among multiple parties and sources, which is typically between multiple web services. This is the basic distinction between orchestration and choreography. Orchestration describes the executable business processes, which interact with both internal and external web services, and the control of the processes is always from the perspective of one of the business parties. Choreography is more collaborative in nature and describes the interactions from the perspective of all parties involved in the process. A detailed description and the differences between the two can be found in [95]. There have been several proposals for describing web services for business processes presented in the recent years including BPML [1] by BPMI, XLANG [113] and BizTalk [73] by Microsoft, WSFL [66] by IBM, BPEL4WS [35] by OASIS (draft standard).

2.2.1 Long running transactions in web services

Web services evolved as a means to integrate processes and applications at an inter-enterprise level. Instead of having traditional transactions, we need a transaction mechanism for web services that supports long-running transactions to describe the loosely coupled activities in web services. Most of the proposals mentioned above use long running transactions to describe the activities.

In the context of web services, business transactions involve interaction and coordination between several services which may belong to different companies. Business transactions need to deal with faults that can arise in any stage of such an environment and this is both difficult and critical. In a long running transaction the usual database approaches, e.g., rollback, are not possible to handle faults. Cancellation of an air plane, or hotel booking, for instance, may lead to a situation which requires the involvement of non-transactional activities with other resources where absolute rollback is not an option. Usually, a long-running transaction interacts with the real world which makes it difficult

to undo the transaction.

In order to recover from faults in long-running transactions, the concept of compensation was introduced. Compensation is defined in [54] as a mechanism to recover from error or change in business plan. Compensation is the act of making amendments, or making up of previously completed task. If a long running transaction fails, appropriate compensations are to be run to compensate for completed parts of the transaction. The concept of transaction was described in [53] where compensation is associated with transactions that correct errors of an already committed transaction. Later the concept of *sagas* was defined in [49] using compensation to describe long running transactions. *Sagas* partition a transaction into a sequence of sub-transactions. Each sub-transaction has an associated compensation. Failure of a sub-transaction in the sequence will execute the associated compensations of the committed sub-transactions.

2.2.2   Formal methods for web services

Several proposals have already been made to describe the composition of web services and some of them are widely used for web services. However, these standards say little or almost nothing about the correct functioning of web services. Formal methods provide a framework, in particular for specification languages and tool support, to address the issues raised by web services (correctness, composition, description). It has already been argued that web services and their interaction can be best described by using process algebras [72]. Recently a few researchers have proposed to use process algebras in web services. A framework has been presented in [41] for the design and verification of web services using process algebra and their tools. The work is based on LOTOS [17] and a two-way mapping is shown between BPEL and the process algebra LOTOS.

Among several business process modelling languages, BPEL4WS is the most common. One of its distinct features is its ability to program fault and compensation handling mechanisms. Recently several researchers have worked on giving a formal semantics to BPELWS. In [99] Qiu *et al.* defined the operational semantics for a subset of BPEL4WS which they called *BPEL*, focusing especially on the fault and compensation handling. Having a similar goal to address the error handling mechanism of web services, a formal semantics has also been defined in [69] using the popular π-calculus, and extended the calculus to include the transactional facilities. cCSP has mechanisms for process coordination as well as for handling compensations and it can be a good candidate to provide a formal framework to model web services. Later, we will model a business transaction to investigate the expressiveness of cCSP. Formalisation of choreography and orchestration of web services by using other methods can be found in [18, 22, 120]. A brief description on the use of process algebra supporting the formalisation of web services can be found in [103]. A survey of several proposals for web service composition can be found in [75].

Atomicity Change to the state by a transaction is atomic, i.e., either all transaction succeeds or none of it happens.

Consistency The state will be consistent after the operation of the transaction. The actions taken for the transaction do not violate any of the integrity constraints of the state.

Isolation Transactions are executed as of no other transaction is executed at the same time. Even though several transaction execute concurrently, it appears to each transaction, T, that other transaction take place either before or after T.

Durability Once a transaction completes successfully (commits), all of its effect will survive any system failure.

FIGURE 2.1: ACID properties of a transaction

2.3 Transaction Processing

Transactional mechanisms have been studied by several communities in computer science. The term transaction is meant to designate a sequence of operations/actions that preserve database consistency. In order to deal with consistency and failure of transactions, ACID (acronym for *atomicity, consistency, isolation* and *durability*, details are in Figure 2.1) transactions are introduced [53]. ACID transactions are often based on locking mechanisms and usual database approaches such as rollback, checkpoint are used to handle faults to provide atomicity to the transaction.

The introduction of web services has led to a new interest in web transactions, in particular long running transaction. The distinctive feature of long running transactions is that the usual transaction properties, i.e., ACID properties are relaxed and instead of performing rollback, the concept of compensation is introduced in case of failure in the transaction. In spite of having a lot of interest in web transactions, researchers have not yet reached a common agreement on a unique notion of long running transaction.

Many researchers are now focusing on connecting formal methods with web transactions. For example, a small set of operators and their operational semantics are presented in [117] to model the transactional behaviour using process algebra. Traditional ACID properties are kept while modelling transaction, however the important concept of long running transaction is not considered. Several other attempts have already been made by many researchers. By using the Join calculus [46], transactional behaviour is described in [20]. Considering Microsoft BizTalk, Bocchi *et al.* define a language πt-calculus [16], an extension of the asynchronous π-calculus [79], and formally specify the transactional behaviour. The πt-calculus includes a transaction construct that contains a compensation handler and a fault manager. In this approach a transaction process remains active as long as its compensation might be required. This doesn't allow for the sequential composition of compensable transactions in which compensations

are composed in reverse order. M.Mazzara *et al.* [69] extended π-calculus to include transactional facilities, and suggested to merge the fault and compensation handling into a general framework of error handling, and defined an operational semantics for the language. They also explained how to program manually the processes of exception handling and nested transactions with compensation handlers in the defined language.

Recently, Laneve and Zavattaro [63] defined a calculus for web transactions called 'webπ' which is an extension of the asynchronous π-calculus with a timed transaction construct. The major aspects considered in webπ are that the processes are interruptible, failure handlers are activated when main processes are interrupted, and time, which is considered in order to deal with latency of web activities or with message losses. A transaction executes either until its termination or until it fails and upon failure the compensation is activated. However, it has a problem similar to that of πt-calculus [16], where compensations of sequentially composed transactions are not preserved in reverse order and it is not possible to get the compensation of a successfully completed process after the failure of a process composed sequentially with the previous one. Using RCCS, a variant of CCS, [37] proposed a formalisation of transactions that distinguishes between reversible and irreversible actions and at the same time incorporates a distributed backtracking mechanism. A brief description of some of the languages developed to model long running transactions for web services will be presented in Section 2.5.

2.4 Formal Semantics

The formal semantics of a programming language is concerned with building mathematical models for the language to serve as a basis for understanding and reasoning about how programs behave. The semantics models the computational meaning of each program. It provides the abstract entities that represent the relevant features of all possible executions and ignores the details that have no relevance to the correctness of implementations.

In order to describe the semantics of a language, here we only focus on the *dynamic semantics*. The other type of semantics is called static semantics, which concerns checking for well-formedness. Dynamic semantics is concerned with the run-time behaviour of a program. A survey of different frameworks for describing the dynamic semantics of programming languages can be found in [123]. Mosses provided an overview with a brief description of different dynamic semantics [83, 86].

There are three main approaches to dynamic semantics:

- *Operational Semantics* - computations are modelled explicitly.

- *Denotational Semantics* - the meaning of a program phrase is modelled by its denotation.

- *Axiomatic Semantics* - describes properties of programs as sets of constraints, and programs as transforming assertions.

2.4.1 Operational Semantics

In an operational framework, the semantics of a program is specified as an abstract machine or transition system, whose computation represents the possible executions of the program. In operational semantics the main concern is how the states are modified during the execution of the statements. Various approaches have been taken for the operational semantics starting from the early 60's [70]. Here our main focus will be on the Structural Operational Semantics (SOS).

Operational semantics are close to intuition and mathematically simple and provide guidelines for language implementation. Operational semantics describe the behaviour of programs in terms of transitions between computation steps, also called configurations. The configurations consists of programs and the associated data which represent the store of data on which the program works. A transition indicates the move from one configuration to another. Behaviours may be represented by means of transition systems. In order to obtain a reasonable notion of semantic equivalence in operational semantics, the popular notion of bisimulation is introduced [76, 78].

The Structural Operational Semantics (SOS) framework was proposed by Plotkin in 1981 [96] in order to provide a simple and direct approach to the semantic description and since then has been exploited in many research works including [6, 77, 87, 121] and lots more. In SOS, computations are modelled as sequences of transitions between states. The transitions for a process depend only on the transitions for its sub processes. The transition relations are defined by a set of axioms and inference rules. As a computation proceeds, phrases of the program are replaced by the values that they have computed. So in the initial state, the program is purely syntactic and in the final state it has been replaced by its computed values. Plotkin's novelty in defining structural operational semantics is the way in which transitions are deduced inductively on the syntactic structure of the language itself. Inductive definitions are given by sets of rules of the form:

$$\frac{\text{Premises}}{\text{Conclusion}} \quad \text{(Side Condition)}$$

which intuitively tells that the conclusion is derived only when the premises are satisfied. In operational semantics definition, the premises are transitions and the conclusion is a transition.

SOS is also referred to as small-step semantics. The alternative approach is called Natural semantics (big-step semantics) [62]. It is actually a special case of SOS, involving initial and finals states but no intermediate states. The purpose of natural

semantics is to describe how the overall results of executions are obtained, whereas in SOS the purpose is to describe how the individual steps of the computations take place. Mosses extended conventional SOS to introduce modularity in SOS which is called Modular SOS (MSOS) [81, 84, 85].

2.4.1.1 Labelled Transition System

The operational semantics of many languages are defined by using the labelled transition systems defined by Plotkin.

Definition 2.1. A *labelled transition system* (LTS) is a quadruple $\langle S, S_0, Act, \{ \xrightarrow{a} \mid a \in Act \} \rangle$ such that:

- S is a set of *states*,
- S_0 is a set of initial states,
- Act is a set of *actions*,
- $\xrightarrow{a} \subseteq S \times S$ is a *transition relation* for every $a \in Act$

The operational semantics of a language can be expressed by using an LTS where the LTS states correspond to the states of programs, the transition relations in the LTS correspond to atomic evolution steps of states of the corresponding programs, and the LTS actions describe the activities of the transitions.

Actions are used as the labels of the transitions. Following the standard notation we write a transition as $s1 \xrightarrow{a} s2$, specifying that there is a transition from state $s1$ to state $s2$ by the action labelled by a.

2.4.2 Denotational Semantics

The framework of denotational semantics was developed by Scott and Strachey [80, 109]. The aim was to provide a mathematics foundation for reasoning about programs and for understanding the fundamental concepts of programming languages. It has then been used in several research including [87, 104, 121].

In the denotational framework the meaning of a program phrase also called denotation is defined abstractly as an element of some suitable mathematical structure, which reflects the contribution of the phrase to the overall program behaviour. A mathematical structure $\mathcal{M}$ constitutes the answer to the question "What does that program phrase actually mean?" A mapping is then defined which associates each program phrase P an element $[\![P]\!]$ of $\mathcal{M}$, called the meaning or denotation of P. The intended behaviour of a program can be easily read from its denotation. The denotational semantics are defined as compositional, in the sense that the denotation of a program is completely determined by the denotation of its constituent subprograms.

2.4.3   Axiomatic Semantics

Axiomatic semantics describe properties of programs as a set of assertions, and programs as transforming assertions. It is particularly suited to proving properties about programs. Axiomatic semantics for sequential programs was developed by Hoare [57] and called Hoare logics. A Hoare logic gives the rules for the relation between assertions about values of variables before and after the execution of each program construct. The constructs concerned are statements S. Suppose that P and Q are assertions about values of particular variables; then $P\{S\}Q$ is the partial correctness formula which states that if P holds at the beginning of the execution of S and the execution of S terminates and Q holds at the end of the execution of S. In partial correctness, the statement S might not terminate. If the execution of S from a state which satisfies P terminates then it is called total correctness. If P does not hold at the beginning then neither S requires to terminate nor does Q need to hold after S. This notion allows us to move from expressions written in programming languages to expressions in logic. The relations $P\{S\}Q$ are specified inductively by rules similar to the transition rules of operational semantics.

2.5   Modelling long running transactions

In this section we put our attention towards formalising long running transactions using process algebras. Some of the recent works on modelling long running transactions are presented here.

StAC

The basic idea of compensation of our current work came from a previous work [24] where `StAC` (Structured Activity Compensation) was introduced as a business process modelling language. The authors believe that compensation gives more flexibility than that of other approaches like rollback. In `StAC` the system description determines the way to connect the components of a system to create a complete system. In this system `StAC` describes the execution order of operations and a B [3] specification is used to describe the states of the system and its basic operations as well. As there are commercial and academic tools (ProB [65]) available for B, the specification in B can be animated and at the same time proof obligations can be obtained. The concept of compensation is extended in StAC_i, where a process can have several independent compensation threads identified by index.

The work on `StAC` was inspired from the CSP process algebra. `StAC` allows sequential and parallel composition of processes and all these operators with compensations operation. `StAC` has early termination that might arise if an exception occurs where the executing process terminates and the remaining tasks may be

abandoned. *Attempt blocks* are used in early termination. An attempt block $P\{Q\}R$ first executes Q, and if Q terminates successfully it then continues with P. If an early termination operation is executed in Q, the block continues with R. Early termination is a useful feature in business processing with the possibility of terminating processes before concluding their main task. Compensations are expressed as pairs of the form $P \div Q$ where Q is the compensation of the process P. `Accept` and `reverse` are other two operators of `StAC`, where `accept` is used to accept a processing clearing its accumulated compensation, and reverse is the operator that causes the accumulated compensation to execute. The operational semantics of `StAC` is presented by using LTS, where transition rules are defined between configurations. The operational semantics of `StAC` can be found in [25, 42].

In [31] an extension of compensation is presented in the context of BPBeans. BPBeans framework allows customers to build Java objects that represent their business processes. The framework also provides for acceptance of tasks and for reversal of tasks. The authors described how StAC extends BPBeans by allowing nested compensation.

Business transactions are very prone to failure in many ways, so any business specification language needs to check the correctness of its specification. The automatic verification of `StAC` specifications using XTL is presented in [8]. Specifications in `StAC` are translated to Prolog equations that can be fed into the XTL model checker. Later we will describe how we use XTL to model check our operational semantics.

Saga Calculus

Bruni *et al* [21] have developed a formal semantics for long running transactions which has compensation handling mechanisms. In the spirit of process description languages (π-calculus or CCS) one of the main goals of this language is to make the distinction between compensation and exception handling. The semantic definition starts with defining sequential activities called *sequential saga* which is a sequence of atomic activities. A sequential saga consists of a process P where each step in P is either a basic activity A or a compensated activity of the form $A \div B$, where A is the normal flow of activity and B is its compensation. Processes which are sequential are represented as $(P \ ; \ Q)$.

The Saga calculus is extended to parallel sagas to support the parallel composition of processes. Initially, only independent parallel branches are defined (Naïve definition) and then extended to full parallel definition. To localise the transactions nesting is introduced in sagas, where a nested transaction is decomposed into a hierarchy of activities called sub-transactions. The invocation of compensation in saga is automatic. The operational semantics of the saga calculus is presented by using big-step semantics.

A comparison has been made in [19] between cCSP and sagas, where the comparison highlights how compensations are handled in concurrent processes when a sibling process in the parallel composition aborts. In cCSP, interruption in parallel processes is coordinated, i.e., when one process throws an interrupt another process catches that interrupt by yielding an interrupt. The compensation procedure is to be activated after the executing processes are stopped. Whereas, in Naïve sagas, all processes in parallel, execute until completion but, if needed they compensate without waiting for processes to complete. On the other hand, in revised sagas, when one parallel process aborts, the other processes execute, but if needed they are interrupted and then their compensations can be executed independently from the rest of the processes. For a detailed discussion please refer to [19].

2.6 Compensating CSP

Compensating CSP (cCSP) is a language used to model long running transactions. The development of the language was inspired by two main ideas: transaction processing features and process algebra, especially, CSP. In this section, we briefly introduce the cCSP language. We describe how compensation constructs are orchestrated to model a long running transaction within the framework of the CSP process algebra.

As in CSP, processes in compensating CSP are modelled in terms of atomic events they can engage in and the operators provided by the language support sequencing, choice, parallel composition of processes. In order to support failed transactions, compensation operators are introduced. The processes are categorised into *standard* and *compensable* processes, where a compensable process has its attached compensation that will execute to compensate the committed actions when required. We use $P, Q, \ldots$ to identify standard processes and $PP, QQ, \ldots$ to identify compensable processes. The syntax of compensating CSP is summarised in Figure 2.2.

The basic unit of a standard process is an atomic event. Standard processes are constructed with the usual CSP operators for choice, sequencing and parallel composition. The process *SKIP* terminates immediately successfully. The language also provides interrupts and interrupt handling. The primitive process *THROW* throws an interrupt immediately. In a purely sequential process, the exception causes an immediate disruption to the flow of control. An interrupt handler may be used to catch interrupts: in $P \rhd Q$, an interrupt raised by P triggers execution of the handler Q. In parallel processes, the whole group of parallel processes may fail when one of the processes throws an exception and all the other processes are willing to disrupt their flow of control and yield to the exception. A process that is ready to terminate is also willing to yield to an interrupt. A process may also yield at mid points in its execution. Yield points are inserted into a process though the primitive *YIELD* process. For example,

Standard Processes:

$$
\begin{aligned}
P, Q ::=\;\; & A && \text{(atomic action)} \\
\mid\;\; & P \; ; \; Q && \text{(sequential composition)} \\
\mid\;\; & P \,\square\, Q && \text{(choice)} \\
\mid\;\; & P \parallel Q && \text{(parallel composition)} \\
\mid\;\; & SKIP && \text{(normal termination)} \\
\mid\;\; & THROW && \text{(throw an interrupt)} \\
\mid\;\; & YIELD && \text{(yield to an interrupt)} \\
\mid\;\; & P \,\triangleright\, Q && \text{(interrupt handler)} \\
\mid\;\; & [PP] && \text{(transaction block)}
\end{aligned}
$$

Compensable Processes:

$$
\begin{aligned}
PP, QQ ::=\;\; & P \div Q && \text{(compensation pair)} \\
\mid\;\; & PP \; ; \; QQ && \\
\mid\;\; & PP \,\square\, QQ && \\
\mid\;\; & PP \parallel QQ && \\
\mid\;\; & SKIPP && \\
\mid\;\; & THROWW && \\
\mid\;\; & YIELDD && \\
\mid\;\; & PP \boxtimes QQ && \text{(speculative choice)}
\end{aligned}
$$

FIGURE 2.2: Syntax of compensating CSP

(P ; $YIELD$; Q) is willing to yield to an interrupt between execution of P and Q. Parallel composition is defined so that throwing of an interrupt in one process synchronises with yielding in another process. The cCSP semantics presented in this chapter does not include synchronised communication between parallel processes. Parallel process groups synchronise only on joint execution of compensation, joint termination and joint interruption. Later in Chapter 5, we define the semantics for synchronisation between processes.

A compensable process is one which has compensation actions attached to it. A compensable process consists of a forward behaviour and a compensation behaviour. In the case of an exception, the compensation will be executed to compensate the forward behaviour. Both the forward and the compensation behaviour are standard processes. The basic way of constructing a compensable process is through the compensation pair construct $P \div Q$, where P is the forward behaviour and Q is its associated compensation. Q should be designed to compensate for the effect of P and may be run long after P has completed. For example, for the atomic events A, A', B and B'

– $(A \div A')$; $(B \div B')$ – behaves as $(A \; ; \; B)$ and has the compensation $(B' \; ; \; A')$, stored for future use.

– $(A \div A')$; $(B \div B')$; $THROWW$ – behaves as A ; B ; B' ; A'. As there is a $THROWW$ at the end, after behaving as before, in this case the compensations are also executed (Here, $THROWW$ is the compensable counterpart of standard basic process $THROW$,

the definition is given later in this section).

The parallel and sequential composition operators for compensable processes are redefined in such a way which ensures that after the failure of a transaction the necessary atomic transactions are performed in an appropriate order to compensate the effect of already performed actions. Sequential composition of compensable processes is defined so that the compensations for all performed actions will be accumulated in the reverse order to their original performance. Parallel composition of compensable processes is defined so that compensations for performed actions will be accumulated in parallel. For example,

$$(A \div A') \parallel (B \div B') \; ; THROWW \quad = \quad (A \parallel B) \; ; (A' \parallel B')$$

By enclosing a compensable process PP in a transaction block $[PP]$ we get a complete transaction which converts the compensable process PP into a standard process. The standard behaviours of the transaction block are defined in terms of the compensable behaviour of PP. Successfully completed PP represents successful completion of the whole transaction block and compensations are no longer needed, and they are discarded. The failed behaviour of PP needs to involve the actual execution of compensations. For example, the following laws shows that compensation will run in the case of exception and discarded for successful termination.

$$[\, A \div A' \; ; THROWW \,] \quad = \quad A \; ; A'$$
$$[\, A \div A' \,] \quad = \quad A$$

The intention of forming a complete transaction from a compensable process is that, in the case of failure, the attached compensations cancel the forward actions and no trace of actions is observable from outside of the transaction block. This satisfies the fundamental principle for a process algebra to model long-running transaction, stating that a transaction either does nothing, because its forward actions will be cancelled, or completes successfully.

A standard process can also be transformed into a compensable process by adding to it a compensation process, which actually does nothing (*SKIP*). The compensable basic processes are defined follows:

$$SKIPP \quad = \quad SKIP \div SKIP$$
$$THROWW \quad = \quad THROW \div SKIP$$
$$YIELDD \quad = \quad YIELD \div SKIP$$

A goal of a transaction can be achieved in different ways and these means can be run in parallel in order to improve responsiveness. The is achieved in speculative choice. In

speculative choice $(PP \boxtimes QQ)$, both processes are allowed to run in parallel. When one attempt succeeds the other attempt can be abandoned and compensations can be used to cancel the effect of the abandoned attempts.

Example:(*Order Fulfilment*)

To illustrate the use of cCSP, we present an example of a transaction for processing customer orders in a warehouse. When the warehouse receives an order from a customer, the first step is to verify whether the stock is available. If not available the customer is informed that the order cannot be accepted. Otherwise, the warehouse starts preparing the order for shipment, and a courier is booked to deliver the goods to the customer. While preparing the order, the warehouse also does a credit check on the customer to verify that the customer can pay for the order. The credit check is performed simultaneously with other tasks because it normally succeeds and the warehouse does not wish to delay the order. If the credit check is failed the processing of the order is stopped. The example is presented in Figure 2.3 in the cCSP language. We present a simple representation of the order acceptance and focus on the order fulfilment part in more detail.

$$
\begin{aligned}
\text{OrderTransaction} \ &= \ [\ \textbf{ProcessOrder}\ ] \\
\textbf{ProcessOrder} \ &= \ (AcceptOrder \div RestockOrder)\,;\ \textbf{FulfillOrder} \\
\textbf{FulfillOrder} \ &= \ BookCourier \div CancelCourier \parallel \\
&\quad\ \ \textbf{PackOrder} \parallel \\
&\quad\ \ CreditCheck\,;\ (\ Ok\,;\ SKIPP \\
&\qquad\qquad\qquad\quad \Box\, NotOk\,;\ THROWW\ ) \\
\textbf{PackOrder} \ &= \ \big\|_{i \in Items} \bullet (PackItem(i) \div UnpackItem(i))
\end{aligned}
$$

FIGURE 2.3: Order transaction example

The first step in the transaction is a compensation pair. The primary action of this pair is to accept the order and deduct the order quantity from the inventory database. The compensation action simply adds the order quantity back to the total in the inventory database. After an order is received from a customer, the order is packed for shipment, and a courier is booked to deliver the goods to the customer. The *PackOrder* process packs each of the items in the order in parallel. Each *PackItem* activity can be compensated by a corresponding *UnpackItem*. Simultaneously with the packing of the order, a credit check is performed on the customer. The credit check is performed in parallel because it normally succeeds, and in this normal case the company does not wish to delay the order unnecessarily. In the case that a credit check fails, an interrupt is thrown causing the transaction to stop its execution, with the courier possibly having been booked and possibly some of the items having being packed. In case of failure,

the semantics of the transaction block will ensure that the appropriate compensation activities will be invoked for those activities that did take place.

2.6.1 Trace Semantics

A trace records the behaviour of a process up to some moment in time. The interactive behaviour of a process can be recorded as a set of traces of all its observable behaviour followed by a special internal terminal action, indicating the way of termination of the process. The traces of composite processes are defined in terms of their constituent processes. In a trace model, each action is considered as atomic. We only present a brief overview of the trace model of cCSP. For a detailed discussion, please refer to [27].

Standard Processes

A process is assumed to have an alphabet of actions Σ which does not include the special terminal events $\Omega = \{\checkmark, !, ?\}$. Standard processes are defined as non-empty set of traces of the form $s\langle\omega\rangle$ where $s \in \Sigma^*$ and $\omega \in \Omega$. All the traces of standard processes are of one of the following forms:

– $s\langle\checkmark\rangle$ trace leading to normal termination

– $s\langle!\rangle$ trace leading to interrupt throw

– $s\langle?\rangle$ trace leading to interrupt yield

A standard process is modelled as a set of such traces. A process yields to catch the interrupt thrown by another communicating process. For traces s and t, we write st as their concatenation. Only completed traces are included in the traces and the nature of a trace is indicated by its final symbol. We first define the operators on traces and then lift them to processes. In the following sections the trace semantics of standard processes are defined as $T(P)$ for standard processes P.

Atomic Action: An atomic event performs an atomic action and terminates successfully.

$$\text{For } A \in \Sigma \qquad T(A) = \{\langle A, \checkmark \rangle\}$$

Basic Processes: Basic processes only have terminal events.

– *SKIP* terminates immediately successfully: $T(SKIP) = \{\langle\checkmark\rangle\}$
– *THROW* throws an interrupt: $T(THROW) = \{\langle!\rangle\}$

– *YIELD* yields an interrupt or terminates: $T(YIELD) = \{\langle\,?\,\rangle, \langle\checkmark\rangle\}$

Choice: The traces of a choice between two processes are determined by the union of their traces.

$$T(P \,\square\, Q) = T(P) \cup T(Q)$$

Sequential Composition: In sequential composition $(P \,;\, Q)$, the traces of Q is will be augmented with the traces of P for a successfully completed P, otherwise the traces of Q is discarded. For processes P and Q, and their traces p and q:

$$
\begin{aligned}
p\langle\checkmark\rangle \,;\, q &= pq \\
p\langle\omega\rangle \,;\, q &= p\langle\omega\rangle \quad \text{where } \omega \neq \checkmark \\
T(P \,;\, Q) &= \{p \,;\, q \mid p \in T(P) \wedge q \in T(Q)\}
\end{aligned}
$$

Interrupt Handler: For processes P and Q, $P \rhd Q$ behaves as P until an interrupt is raised by P.

$$
\begin{aligned}
p\langle\,!\,\rangle \rhd q &= pq \\
p\langle\omega\rangle \rhd q &= p \quad \text{where } \omega \neq\ ! \\
T(P \rhd Q) &= \{p \rhd q \mid p \in T(P) \wedge q \in T(Q)\}
\end{aligned}
$$

Parallel Composition: Parallel composition defined in [27] does not consider the synchronisation of observable events. Processes are considered to synchronise on joint termination or interruption. The synchronisation of terminal events is denoted by defining a synchronisation operator on terminal events from the set Ω. If ω and ω' are terminal events from distinct parallel processes, the joint terminal event by their concurrent execution is denoted by $\omega\&\omega'$. Table 2.1 enumerates the evaluation of the operator. The first three rows of the table show that synchronization of any terminal event with an interrupt throw results in an interrupt throw. The next two rows state that synchronization between a yield with either a yield or a successful termination results a yield. The last row of the table shows that composition of two parallel processes will terminate successfully when both processes terminate successfully.

TABLE 2.1: Synchronisation of terminal events

ω	ω'	$\omega\&\omega'$
!	!	!
!	?	!
!	$\checkmark$	!
?	?	?
?	$\checkmark$	?
$\checkmark$	$\checkmark$	$\checkmark$

The execution of actions in separate processes occurs in an interleaving fashion. Asyn-

chronous execution can lead to different interleavings: $A \parallel B$ can execute A followed by B or B followed by A. For traces p, q we write $p \parallel\!\mid q$ to denote their interleaving:

$$\langle x \rangle p \parallel\!\mid \langle y \rangle q \quad = \quad \{ \langle x \rangle r \mid r \in (p \parallel\!\mid \langle y \rangle q \} \; \cup \; \{ \langle y \rangle r \mid r \in (\langle x \rangle p \parallel\!\mid q) \}$$

The parallel composition of traces is defined to be the set of interleavings of their observable events followed by the synchronisation of their terminal events.

$$p \langle \omega \rangle \parallel q \langle \omega' \rangle \quad = \quad \{ r \langle \omega \, \& \, \omega' \rangle \mid r \in (p \parallel\!\mid q) \}$$

$$T(P \parallel Q) \quad = \quad \{ r \mid r \in (p \parallel q) \wedge p \in T(P) \wedge q \in t(Q) \}$$

Compensable Processes

A compensable process has both forward and compensation behaviour. The compensable behaviour is modelled by a pair of traces of the form $(p \langle \omega \rangle, p' \langle \omega' \rangle)$ where $p \langle \omega \rangle$ is for forward behaviour and $p' \langle \omega' \rangle$ is for the compensation.

Choice: The choice of compensable processes are same as standard processes which is determined by the union of their traces.

$$T(PP \,\square\, QQ) \; = \; T(PP) \cup T(QQ)$$

Sequential Composition: The trace semantics are defined in such a way that compensations are accumulated in reverse to the forward processes.

$$
\begin{aligned}
(p \langle \checkmark \rangle, p') \,;\, (q, q') \quad &= \quad (pq, \; q'; p') \\
(p \langle \omega \rangle, p') \,;\, (q, q') \quad &= \quad (p \langle \omega \rangle, p'), \quad \text{where } \omega \neq \checkmark \\
T(PP \,;\, QQ) \quad &= \quad \{ pp \,;\, qq \mid pp \in T(PP) \wedge qq \in T(QQ) \}
\end{aligned}
$$

Parallel Composition: The trace semantics of compensable parallel composition is similar to the standard case.

$$
\begin{aligned}
(p, p') \parallel (q, q') \quad &= \quad \{ (r, r') \mid r \in (p \parallel q) \wedge r' \in (p' \parallel q') \} \\
T(PP \parallel QQ) \quad &= \quad \{ rr \mid rr \in (pp \parallel qq) \wedge pp \in T(PP) \wedge qq \in T(QQ) \}
\end{aligned}
$$

Compensation Pair: The pairing operator is first defined on traces and then lifted

to processes.

$$p\langle\checkmark\rangle \div q \;\; = \;\; (p\langle\checkmark\rangle, q)$$
$$p\langle\omega\rangle \div q \;\; = \;\; (p\langle\omega\rangle, \langle\checkmark\rangle), \quad \text{where } \omega \neq \checkmark$$
$$T(P \div Q) \;\; = \;\; \{\, p \div q \mid p \in T(P) \land q \in T(Q) \,\}$$

The trace semantics defined in [27] has an extra behaviour that allows the compensation pair to yield immediately with empty compensation. In this thesis we only use the simplified version of the pair operator presented above. We can get the same behaviour as in the original definition by adding a yield to the pair operator as follows:

$$P \div' Q \;=\; YIELD \; ; \; P \div Q$$

Speculative Choice: In speculative choice ($PP \boxtimes QQ$), the forward behaviour of either processes can terminate successfully, or the forward behaviour of both processes can terminate successfully, or neither of them terminate successfully. The trace semantics of this operator is defined as follows:

$$(p\langle\checkmark\rangle, p') \boxtimes (q\langle\omega\rangle, q') \;\; = \;\; \{\, (rq', p') \mid r \in (p \,|||\, q) \,\} \quad (\omega \neq \checkmark)$$
$$(p\langle\omega\rangle, p') \boxtimes (q\langle\checkmark\rangle, q') \;\; = \;\; \{\, (rp', q') \mid r \in (p \,|||\, q) \,\} \quad (\omega \neq \checkmark)$$
$$(p\langle\checkmark\rangle, p') \boxtimes (q\langle\checkmark\rangle, q') \;\; = \;\; \{\, (rq', p') \mid r \in (p \,|||\, q) \,\} \cup$$
$$\{\, (rp', q') \mid r \in (p \,|||\, q) \,\}$$
$$(p\langle\omega\rangle, p') \boxtimes (q\langle\omega'\rangle, q') \;\; = \;\; \{\, (rr'), \langle\checkmark\rangle \mid r \in (p \,|||\, q) \land r' \in (p' \,\|\, q') \,\} \; (\omega, \omega' \neq \checkmark)$$
$$T(PP \boxtimes QQ) \;\; = \;\; \{\, pp \boxtimes qq \mid pp \in T(PP) \land qq \in T(QQ) \,\}$$

Transaction Block: A transaction block runs the compensation for an interrupted forward behaviour and discards the compensation for successfully terminating forward behaviour and removes the traces whose forward behaviour is yielding.

$$[\,p\langle!\rangle, p'\,] \;\; = \;\; (pp')$$
$$[\,p\langle\checkmark\rangle, p'\,] \;\; = \;\; p\langle\checkmark\rangle$$
$$T([\,PP\,]) \;\; = \;\; \{[\,p, p'\,] \mid (p, p') \in T(PP)\}$$

If PP is empty, then the transaction block will be empty. The healthiness condition that ensures that the set of traces of a transaction block is non-empty is denoted by stating that all processes P and PP consist of some terminating behaviour:

– $p\langle\checkmark\rangle \in T(P)$ or $p\langle!\rangle \in T(P)$, for some p

– $(p\langle\checkmark\rangle, p') \in T(PP)$ or $(p\langle!\rangle, p') \in T(PP)$, for some p, p'

2.7 Prototype Verification System (PVS)

PVS (Prototype Verification System) [89] is a general purpose proof tool that provides an interactive environment for writing formal specifications and checking formal proofs. It consists of an expressive specification language that comes with a *type checker*, *theorem prover* and other related tools. Details of PVS can be found in the PVS system guide [91]. Several research have been carried out to mechanise the trace semantics of standard CSP by using the theorem prover PVS, e.g., [38, 40], which give us a good starting point to use PVS in our experiments to model the semantic models of cCSP. We also use PVS to mechanise the relationship between the semantic models.

2.7.1 PVS Specification Language

PVS has an expressive specification language augmented with classical higher order logic. A typical specification consists of a collection of modular units called *theories*. Theories are imported to other theories making a hierarchy of theories. The definitions and theories in the imported theories become available to the importing theories. Theories are identified by their names and can be parameterised by a list of identifiers.

Types

The PVS specification language has a very sophisticated type system. Simple types are constructed from available base types, such as, boolean (bool), natural (nat) etc. by using function [···->···], tuple [···,···]. Constructor, such as sets and predicates, are represented as boolean functions, where functions return boolean values. The term 'set', 'predicate' and 'boolean function' are used interchangeably in this context. There is also support for records, sequences, lists and trees, etc.

The type system in PVS is even enriched by supporting uninterpreted type declarations, predicate subtypes, dependent types, enumerated types, and a mechanism for defining abstract datatypes such as lists, trees. Subtyping makes type checking more powerful. However, this introduces undecidability in type checking: if PVS is unable to check the type of an expression then it generates one or more 'type correctness condition's (TCC). These must be proved to consider that the expression is type checked. Many TCCs are discharged automatically, whereas more involved ones require to use the interactive prover.

2.7.2 PVS Theorem Prover

The PVS prover provides the primitives to perform inductive reasoning, rewriting and model checking. The prover is interactive and uses sequent style presentation. The

prover provides powerful basic commands and a mechanism for building reusable strategies based on these. The prover maintains a *proof tree* and to prove the tree the nodes of the tree have to be proved using the proof commands provided by PVS. A node in the tree is proved if its leaves are proved. The nodes of the proof tree are called *proof goals*. Each proof goal is a sequent consisting of a list of antecedent formulae followed by a list of consequent formulae separated by '|----'. Each of the proof goals are tackled separately and a single proof goal is displayed with the `Rule?` prompt. A typical goal can be as follows:

```
example_theorem.x:
[-1] A
[-2] B
[-3] C
|-------
[1] S
{2} T
Rule?
```

where `example_theorem` is the formula name and `x` identifies the current goal. `A, B, C` are the antecedent formula and `S,T` are the consequent. Intuitively, the above sequent can be interpreted as the conjunction of the antecedents implies the disjunction of the consequents, i.e., $A \wedge B \wedge C \Rightarrow S \vee T$. The proof tree starts with a single root node consisting of the theorem to be proved. A proof tree is built by adding subtrees to leaf nodes directed by the proof commands. Each formula is uniquely numbered where antecedents are labelled with a negative number and consequents are labelled by positive numbers. The formula numbers in braces indicate those formula that are either new or different from those in parent sequent whereas the formula numbers in square brackets indicate formulas that are unchanged in a subgoal from the parent goal.

The prover accepts commands in Emacs via a Lisp-like interface. The commands consist of high level commands called strategies and more specific commands called rules. Strategies are used to tackle a broad range of problems and can finish proofs automatically. The user gets much control over the proofs using the rules. The PVS prover guide [90] gives a detailed description of the theorem prover as well as the available proof commands. In order to be able to use the PVS theorem prover it is necessary to understand the available proof commands. We will use the theorem prover in order to prove the theorems and lemmas that will be defined to show the correspondence between the semantic models of cCSP.

Chapter 3

Operational Semantics for cCSP

According to the objectives laid down in Chapter 1, we wish to extend the existing semantic models of compensating CSP. In this chapter we define an operational semantics for cCSP. We also show the encoding of the operational semantics for both standard and compensable processes into Prolog.

3.1   Introduction

Compensating CSP (cCSP) is a language, introduced by Butler *et al* [27], for modelling long running business transactions . The language is a variant of standard CSP process algebra [59] with constructs for orchestration of compensations. With the introduction of the language, its formal semantics is defined by giving the denotational semantics (trace) semantics. Although the denotational semantics has its own advantages, e.g., inherently compositional, the operational semantics has been widely accepted amongst language developers and practitioners as it is easy to understand and close to implementations.

Operational semantics describes the behaviour of programs in terms of transitions between program states or configurations. The overall state of the program is divided into a number of components. The transition shows the move of the program from one configuration to another. In labelled transition systems (LTS), the transitions are labelled by the information on the activity performed by the transition. An operational semantics is given by a set of rules which specify how the states of a program change during execution. Each rule specifies a certain precondition on the content of some component and their new content after the application of the rule.

Structural operational semantics (SOS) [87, 97, 98] was introduced by Plotkin [96] as a logical means to define operational semantics. The basic idea behind SOS is to define the behaviour of a program in terms of the behaviour of its parts, thus providing a

structural view on operational semantics. Due to its intuitive look and easy to follow structure, SOS has become widely popular in defining operational semantics.

3.1.1 Chapter Contribution

This chapter presents the operational semantics of cCSP. The semantics is presented by using a labelled transition system following the approach of Plotkin [96]. The operational semantics gives a precise understanding of the execution of the language. The semantics gives a one-state-at-a-time recipe for computing the transition system of any process. Operational semantics for standard CSP is defined in [47] and later in [102] and having similarity with CSP, our work builds on that.

We make the operational semantics executable by directly encoding the rules in Prolog. Our hope is that this can serve as a useful basis for model checking cCSP processes. XTL [8] is a model checker which allows a wide range of system specifications. It accepts specifications written using high-level Prolog predicates describing the transition between different states of the system. Given a Prolog encoding of the operational semantics, the XTL package provides us with an experimental animator and model checker for cCSP.

3.1.2 Chapter Structure

This chapter is organised as follows. Section 3.2 presents the operational semantics of cCSP. The encoding of operational semantics in XTL is presented in Section 3.3. After giving some related works in Section 3.4 we draw the conclusions in Section 3.5.

3.2 Operational Semantics

The operational semantics is a way of defining the behaviour of processes by specifying atomic transitions on process terms. It describes how the individual steps of the computation take place and therefore offers a direct intuition of how program constructs are intended to behave. Inference rules are presented which define the transitions that a process may perform, which for composite programs are given in terms of the possible transition of the constituents. Side conditions are used in the inference rules which state that the deduction is valid only under the stated restriction. Labelled transition systems (LTS) [96] are used to define the transitions of process terms. Separate rules are defined for each operator.

It is mentioned earlier that the alphabet of events is separated into observable (normal) and terminal events. The set of observable events is represented by Σ. Normal events

result in the transition of a process from one state to another. For example, the normal event a makes the transition of a standard process from P to P' and a compensable process from PP to PP'.

$$P \xrightarrow{a} P' \quad (P' \text{ is a standard process})$$
$$PP \xrightarrow{a} PP' \quad (PP' \text{ is a compensable process})$$

The terminal events $\Omega = \{\checkmark, !, ?\}$ represent the different ways in which a process can terminate: successful termination is represented by the $\checkmark$ event, termination by throwing an interrupt is represented by ! and yielding an interrupt is denoted by ?. In order to define the operational semantics, the language terms are extended with a null process (0) that cannot perform any event. The terminal events act differently on standard and compensable processes. When a standard process performs a terminal event ω ($\omega \in \Omega$) the process terminates either normally or abnormally and no further operation occurs.

$$P \xrightarrow{\omega} 0 \quad (\omega \in \Omega)$$

When a compensable process PP executes a terminal event, instead of evolving to the null process (0), it evolves to a standard process P representing its compensation.

$$PP \xrightarrow{\omega} P \quad (\omega \in \Omega)$$

In Section 3.2.2 we will see how these resulting compensations are treated by the various operators for compensable processes.

3.2.1 Semantics of Standard Processes

This section presents the operational semantics of standard processes of compensating CSP. Transition rules are defined for observable and terminal events. Recall that transition rules are different for these two kind of events. In order to represent the labels in the transition rules, variable a is used to range over the set Σ of observable transitions, and ω to range over Ω for terminal transitions. There are some transition rules which are common to both observable and terminal events and the variable α ranges over the combined set $\Sigma \cup \Omega$.

Atomic Action

A process A performs an atomic action A and then terminates successfully:

$$A \xrightarrow{A} SKIP \quad (A \in \Sigma) \tag{3.1}$$

Basic Processes

SKIP, *THROW* and *YIELD* are the primitive processes of cCSP. These processes have only terminal events and the effect of terminal events on these basic processes are presented here:

$$
\begin{aligned}
SKIP &\xrightarrow{\checkmark} 0 \\
THROW &\xrightarrow{!} 0 \\
YIELD &\xrightarrow{?} 0 \\
YIELD &\xrightarrow{\checkmark} 0
\end{aligned}
\tag{3.2}
$$

Choice

Occurrence of an event, observable or terminal, in either process, in a choice operation, resolves the choice:

$$
\frac{P \xrightarrow{\alpha} P'}{P \ \square \ Q \xrightarrow{\alpha} P'} \quad \frac{Q \xrightarrow{\alpha} Q'}{P \ \square \ Q \xrightarrow{\alpha} Q'} \quad (\alpha \in \Sigma \cup \Omega)
\tag{3.3}
$$

Sequential Composition

In a sequential composition $(P \ ; \ Q)$, P may perform any non-terminal event and Q remains as it is:

$$
\frac{P \xrightarrow{a} P'}{(P \ ; \ Q) \xrightarrow{a} (P' \ ; \ Q)} \quad (a \in \Sigma)
\tag{3.4}
$$

When P terminates successfully with a $\checkmark$ then Q can start to execute and the $\checkmark$ from P is hidden. Here the resulting event is α which can be either a normal or a terminal event and the resultant process is dependant on α.

$$
\frac{P \xrightarrow{\checkmark} 0 \wedge Q \xrightarrow{\alpha} Q'}{(P \ ; \ Q) \xrightarrow{\alpha} Q'} \quad (\alpha \in \Sigma \cup \Omega)
\tag{3.5}
$$

When P terminates by throwing or yielding an interrupt the whole sequential composition is terminated:

$$
\frac{P \xrightarrow{\omega} 0}{(P \ ; \ Q) \xrightarrow{\omega} 0} \quad (\omega \in \{!, ?\})
\tag{3.6}
$$

Interrupt Handler

The interrupt handler is similar to sequential composition, except that the flow of control from the first to the second process is caused by the throw event rather than the $\checkmark$ event. For processes P and Q, $P \rhd Q$ represents a process that behaves as P until an interrupt is raised by P, at which point it behaves as Q.

The process P in $P \rhd Q$ can evolve to another state by any normal event:

$$\frac{P \xrightarrow{a} P'}{P \rhd Q \xrightarrow{a} P' \rhd Q} \quad (a \in \Sigma) \tag{3.7}$$

When an interrupt is raised by P, the interrupt handler Q starts to execute:

$$\frac{P \xrightarrow{!} 0 \wedge Q \xrightarrow{\alpha} Q'}{P \rhd Q \xrightarrow{\alpha} Q'} \quad (\alpha \in \Sigma \cup \Omega) \tag{3.8}$$

The interrupt handler terminates by a terminal event, except !:

$$\frac{P \xrightarrow{\omega} 0}{P \rhd Q \xrightarrow{\omega} 0} \quad (\omega \in \{\checkmark, ?\}) \tag{3.9}$$

Parallel Composition

Processes in parallel will synchronise on joint interruption or joint termination. For observable events, processes interleave with each other. We extend this definition in Chapter 5 where the processes synchronise over a set of observable events.

Asynchronous execution of the parallel composition $(P \parallel Q)$ occurs in an interleaved way, where the execution of $P \parallel Q$ can be either P followed by Q or Q followed by P:

$$\frac{P \xrightarrow{a} P'}{P \parallel Q \xrightarrow{a} P' \parallel Q} \quad \frac{Q \xrightarrow{a} Q'}{P \parallel Q \xrightarrow{a} P \parallel Q'} \quad (a \in \Sigma) \tag{3.10}$$

Synchronisation of standard processes for terminal events is defined as follows:

$$\frac{P \xrightarrow{\omega} 0 \wedge Q \xrightarrow{\omega'} 0}{P \parallel Q \xrightarrow{\omega \& \omega'} 0} \quad (\omega, \omega' \in \Omega) \tag{3.11}$$

Although the transaction block is a standard process, its transition rules need a description of the compensable process. Hence the transition rules are presented later with the transition rules of compensable processes.

3.2.2   Semantics of Compensable Processes

In this section, we present the semantics of the operators for compensable processes. Recall that a compensable process consists of forward behaviour and compensation behaviour. The compensation can be executed to compensate for the forward action, if necessary.

Choice

In a choice operation $(PP \mathbin{\square} QQ)$, an observable event in either PP or QQ resolves the choice as in standard processes:

$$\frac{PP \xrightarrow{a} PP'}{PP \mathbin{\square} QQ \xrightarrow{a} PP'} \quad \frac{QQ \xrightarrow{a} QQ'}{PP \mathbin{\square} QQ \xrightarrow{a} QQ'} \quad (a \in \Sigma) \tag{3.12}$$

Terminal events $(\omega \in \{\checkmark, !, ?\})$ also resolve the choice but result in the corresponding compensation:

$$\frac{PP \xrightarrow{\omega} P}{PP \square QQ \xrightarrow{\omega} P} \quad \frac{QQ \xrightarrow{\omega} Q}{PP \square QQ \xrightarrow{\omega} Q} \quad (\omega \in \Omega) \tag{3.13}$$

Sequential Composition

The sequential operator is defined in such a way that the compensation of the first process will accumulate after the compensation of the second one.

In a sequential composition of compensable processes $(PP \mathbin{;} QQ)$, as for the standard case, PP may perform non-terminal events while QQ is preserved:

$$\frac{PP \xrightarrow{a} PP'}{PP \mathbin{;} QQ \xrightarrow{a} PP' \mathbin{;} QQ} \quad (a \in \Sigma) \tag{3.14}$$

If PP throws or yields an interrupt, the whole process terminates leaving the compensation from PP:

$$\frac{PP \xrightarrow{\omega} P}{PP \mathbin{;} QQ \xrightarrow{\omega} P} \quad (\omega \in \Omega \wedge \omega \neq \checkmark) \tag{3.15}$$

If PP terminates normally, QQ commences and the compensation from PP should be maintained to be composed with the compensation from QQ at a later stage. In order to deal with this, we introduce a new auxiliary construct to the language of the form $\langle QQ, P \rangle$. The effect of $\langle QQ, P \rangle$ is to execute the forward behaviour of QQ and then

compose the compensation from QQ with P. This is used to define the transfer of control in a sequential composition:

$$\frac{PP \xrightarrow{\checkmark} P \wedge QQ \xrightarrow{a} QQ'}{PP \ ; \ QQ \xrightarrow{a} \langle QQ', P \rangle} \quad (a \in \Sigma) \tag{3.16}$$

However, if QQ can execute a terminal event after PP terminates normally, then instead of introducing the new auxiliary construct, the maintained compensations of both processes are composed in reverse order:

$$\frac{PP \xrightarrow{\checkmark} P \wedge QQ \xrightarrow{\omega} Q}{PP \ ; \ QQ \xrightarrow{\omega} Q \ ; \ P} \quad (\omega \in \Omega) \tag{3.17}$$

The process QQ in the construct $\langle QQ, P \rangle$ can perform non-terminating events:

$$\frac{QQ \xrightarrow{a} QQ'}{\langle QQ, P \rangle \xrightarrow{a} \langle QQ', P \rangle} \quad (a \in \Sigma) \tag{3.18}$$

When QQ terminates then its compensation is composed in front of the existing compensation, which ensures that the compensations are accumulated in reverse order to their original sequential operation:

$$\frac{QQ \xrightarrow{\omega} Q}{\langle QQ, P \rangle \xrightarrow{\omega} Q \ ; \ P} \quad (\omega \in \Omega) \tag{3.19}$$

Parallel Composition

Parallel processes interleave with each other for observable events:

$$\frac{PP \xrightarrow{a} PP'}{PP \parallel QQ \xrightarrow{a} PP' \parallel QQ} \quad \frac{QQ \xrightarrow{a} QQ'}{PP \parallel QQ \xrightarrow{a} PP \parallel QQ'} \quad (a \in \Sigma) \tag{3.20}$$

However, as the processes are compensable, when they synchronise over any terminal events, the forward processes are terminated and the composition is defined in such a way that the corresponding compensation processes will be accumulated in parallel:

$$\frac{PP \xrightarrow{\omega} P \wedge QQ \xrightarrow{\omega'} Q}{PP \parallel QQ \xrightarrow{\omega \& \omega'} P \parallel Q} \quad (\omega, \omega' \in \Omega) \tag{3.21}$$

Compensation Pair

The compensation pair $(P \div Q)$ is constructed from two standard processes. The first one (P) is called the forward process which is executed during normal execution and

the second one (Q) is called the compensation of the forward process which is stored for future use upon successful completion of the forward behaviour. Interrupted forward behaviour results in empty compensations. The rationale of this definition is that compensation is intended to be used at a later stage to compensate a successfully completed forward behaviour and not an interrupted behaviour.

If the forward process can perform a non-terminal event, then so can the pair:

$$\frac{P \xrightarrow{a} P'}{(P \div Q) \xrightarrow{a} (P' \div Q)} \quad (a \in \Sigma) \tag{3.22}$$

If the forward process terminates normally, then the pair terminates with Q as the resulting compensation:

$$\frac{P \xrightarrow{\checkmark} 0}{(P \div Q) \xrightarrow{\checkmark} Q} \tag{3.23}$$

If the forward process terminates abnormally, then so does the pair, resulting in an empty compensation process:

$$\frac{P \xrightarrow{\omega} 0}{(P \div Q) \xrightarrow{\omega} SKIP} \quad (\omega \neq \checkmark) \tag{3.24}$$

The definition of the compensation pair defined in the traces model of cCSP [27] has a subtle difference to that presented here. An extra behaviour for the compensation pair was included in the traces model definition which allows the compensation pair to yield immediately with an empty compensation. This forces an automatic yield at the beginning of the compensation pair. The same behaviour can be obtained using the definition presented here by adding a yield sequentially followed by the forward process.

$$P \div' Q \mathrel{\hat{=}} (YIELD \mathrel{;} P) \div Q$$

YIELD can either yield or terminate with a $\checkmark$. When it yields, the above definition gives us the extra behaviour of yield with an empty compensation and when it terminates by a $\checkmark$ we get the same behaviour presented here.

Compensable Basic Processes

Compensable basic processes are made from standard basic processes by pairing them with *SKIP*, which does nothing. We do not include the operational semantics of these

basic processes here as the semantics can be derived easily by using the pairing operator.

$$
\begin{aligned}
SKIPP &= SKIP \div SKIP \\
THROWW &= THROW \div SKIP \\
YIELDD &= YIELD \div SKIP
\end{aligned}
\qquad (3.25)
$$

The operators on compensable processes are defined in such a way that ensures that the correct compensations will accumulate in the case of an interruption. For example, assume that A and B are atomic and non-yielding, and A' and B' are the attached respective compensations. The following derivation shows how compensation is accumulated:

$$
\begin{aligned}
& A \div A' \; ; \; B \div B' \\
\xrightarrow{A} & \quad SKIP \div A' \; ; \; B \div B' \quad \text{(rule 3.1)} \\
\xrightarrow{\checkmark} & \quad \langle B \div B', A' \rangle \\
\xrightarrow{B} & \quad \langle SKIP \div B', A' \rangle \quad \text{(rule 3.16)} \\
\xrightarrow{\checkmark} & \quad B' \; ; \; A' \quad \text{(rule 3.19)}
\end{aligned}
$$

Transaction Block

Although a transaction block is a standard process rather than a compensable process, we describe its semantics in this section since it requires an understanding of the semantics of compensable processes. A transaction block is formed from a compensable process PP by enclosing PP in a transaction block $[PP]$. A transaction block converts a compensable process into a standard process.

An observable event changes the state of the process inside the block:

$$
\frac{PP \xrightarrow{a} PP'}{[PP] \xrightarrow{a} [PP']} \quad (a \in \Sigma) \qquad (3.26)
$$

Successful completion of the forward behaviour of the compensable process of a transaction block represents successful completion of the whole block and compensation is no longer needed and it is discarded:

$$
\frac{PP \xrightarrow{\checkmark} P}{[PP] \xrightarrow{\checkmark} 0} \qquad (3.27)
$$

When the forward behaviour throws an exception, then the resulting compensation is

run:

$$\frac{PP \xrightarrow{!} P \wedge P \xrightarrow{\alpha} P'}{[PP] \xrightarrow{\alpha} P'} \quad (\alpha \in \Sigma \cup \Omega) \tag{3.28}$$

Since a transaction block is a standard process, P' in this rule is not a compensation that is stored for later execution, rather it describes the behaviour of $[PP]$ after the execution of event α.

Note that there is no rule for a yield transition (?) in a transaction block. This is because a transaction block does not yield to interrupts from the outside. Yields by a sub-process of PP will synchronise with interrupts from some other sub-process resulting in the ! event making yields within PP non-observable.

The following derivation shows how the transition rules can be applied to a transaction block:

$$[A \div A' \; ; \; THROWW]$$
$$\xrightarrow{A} \quad [SKIP \div A' \; ; \; THROWW] \qquad \text{(Rule 3.26)}$$

$$SKIP \div A' \; ; \; THROWW$$
$$\xrightarrow{!} \quad SKIP \; ; \; A' \qquad \text{(Rule 3.23, 3.4)}$$

$$SKIP \; ; \; A'$$
$$\xrightarrow{A'} \quad SKIP \qquad \text{(Rule 3.5)}$$
$$SKIP$$
$$\xrightarrow{\checkmark} \quad 0 \qquad \text{(Rule 3.1)}$$

3.2.3 Speculative Choice

Apart from the operators, for which we have defined operational semantics in this section, the trace semantics for speculative choice is defined in [27]. The speculative choice of processes PP and QQ is written as $PP \boxtimes QQ$. In speculative choice, the forward behaviour of PP and QQ run in parallel until one of them terminates successfully. If one process terminates successfully, the compensation from the other process is to be run and preserve the compensation of the successfully completed process. Here, we outline the operational semantics for the speculative choice operator.

In the speculative choice $PP \boxtimes QQ$, either process can make a transition by an observable

(normal) event, similar to processes in a parallel composition:

$$\frac{PP \stackrel{a}{\longrightarrow} PP'}{PP \boxtimes QQ \stackrel{a}{\longrightarrow} PP' \boxtimes QQ} \quad (a \in \Sigma) \qquad \frac{QQ \stackrel{a}{\longrightarrow} QQ'}{PP \boxtimes QQ \stackrel{a}{\longrightarrow} PP \boxtimes QQ'} \quad (a \in \Sigma)$$

When one process terminates successfully, the compensation of the other process is to be run preserving the compensation of the successfully terminating process:

$$\frac{PP \stackrel{\checkmark}{\longrightarrow} P \wedge QQ \stackrel{\omega}{\longrightarrow} Q}{PP \boxtimes QQ \stackrel{\checkmark}{\longrightarrow} \langle Q, P \rangle} \quad (\omega \neq \checkmark) \qquad \frac{QQ \stackrel{\checkmark}{\longrightarrow} Q \wedge PP \stackrel{\omega}{\longrightarrow} P}{PP \boxtimes QQ \stackrel{\checkmark}{\longrightarrow} \langle P, Q \rangle} \quad (\omega \neq \checkmark)$$

We introduce an auxiliary construct ($\langle Q, P \rangle$, or $\langle P, Q \rangle$) here in order to represent the scenario when one process terminates with a $\checkmark$, and the other process terminates immediately and its compensation is to be run. For example, in the above rule, when the process PP terminates successfully then the process QQ terminates and its compensation Q is to be run whereas the compensation P from PP is to be preserved for future reference. We represent this scenario by using the construct $\langle Q, P \rangle$.

The auxiliary construct can make a transition by an observable event:

$$\frac{P \stackrel{a}{\longrightarrow} P'}{\langle P, Q \rangle \stackrel{a}{\longrightarrow} \langle P', Q \rangle} \quad (a \in \Sigma)$$

When the auxiliary construct makes a terminal transition, the compensation of the successfully terminated process is preserved:

$$\frac{P \stackrel{\omega}{\longrightarrow} 0}{\langle P, Q \rangle \stackrel{\omega}{\longrightarrow} Q} \quad (\omega \in \Omega)$$

When both processes, PP and QQ terminate successfully, the compensation from one of the processes will run and the compensation from the other process will be preserved. This arises a choice between the auxiliary constructs, as shown below:

$$\frac{PP \stackrel{\checkmark}{\longrightarrow} P \wedge QQ \stackrel{\checkmark}{\longrightarrow} Q}{PP \boxtimes QQ \stackrel{\checkmark}{\longrightarrow} \langle Q, P \rangle \square \langle P, Q \rangle}$$

When neither process terminates successfully, compensations from both processes run in parallel and no compensation will be preserved:

$$\frac{PP \stackrel{\omega}{\longrightarrow} P \wedge QQ \stackrel{\omega'}{\longrightarrow} Q}{PP \boxtimes QQ \stackrel{\omega \& \omega'}{\longrightarrow} \langle (P \parallel Q), SKIP \rangle} \quad (\omega, \omega' \neq \checkmark)$$

In the rest of the thesis, we take a subset of the original cCSP operators, leaving speculative choice operator for our future work. The experiment for speculative choice requires a substantial time and having successfully completed experiment for other operators, we

will be in a better position to carry our experiment with speculative choice.

3.3 Prolog Implementation

In this section we outline a Prolog implementation of the operational semantics presented in Section 3.2. We encode the operational rules as Prolog clauses and we use a tool which can animate this encoded semantics. XTL [8] is a model checker which allows a wide range of system specification. It accepts specifications written by using high level Prolog predicates describing the transitions between different states of the system. The XTL animator supports step by step animation showing transitions between different states of the specification and also supports backtracking.

The input language for XTL is very simple. The predicate that is used in XTL is: `trans/3`

`trans(A,S1,S2)`: A transition from state S1 to state S2 by the action A.

Consider the following sample specification:

`trans(a1,p,q). trans(a2,q,p). trans(a3,r,r).`

The above line specifies that by the action `a1`, there is a transition from `p` to `q`, that action `a2` causes the reverse transition and action `a3` causes `r` to `r`.

As the operational semantics of compensating CSP is described by using operational rules, they are easily transferable to corresponding `trans/3` predicates. We reproduce some operational rules and their corresponding Prolog predicates. For example, consider one of the rules for sequential composition of standard processes:

$$\frac{P \xrightarrow{a} P'}{P \; ; Q \xrightarrow{a} P' \; ; Q} \qquad (a \in \Sigma)$$

The Prolog representation of this is:

```
trans(seq(P,Q),A,seq(P1,Q)):-
  NOT(terminal(A)),
  trans(P,A,P1).
```

If we consider a transition by a terminal event then

$$\frac{P \xrightarrow{\omega} 0}{P \; ; Q \xrightarrow{\omega} 0} \; (\omega \neq \checkmark)$$

the above rule can be represented as follows:

```
trans(seq(P,Q),A,0):-
    terminal(A), NOT(A = tick),
    trans(P,A,0).
```

Compensable processes are encoded in a similar way with the compensable operators being differentiated from the standard ones. For example, consider the following rule for compensable sequential composition:

$$\frac{PP \stackrel{a}{\longrightarrow} PP'}{PP \; ; QQ \stackrel{a}{\longrightarrow} PP' \; ; QQ} \quad (a \in \Sigma)$$

This is represented in Prolog as:

```
trans(cseq(PP,QQ),A,cseq(PP1,QQ)):-
    NOT(terminal(A)),
    trans(PP,A,PP1).
```

The XTL package provides us with an experimental animator for cCSP specifications. The tool has the facility to observe step by step animation of the specifications. It allows to see how the rules of the operational semantics have derived each action of a compound process from the action of its parts. The animation helps to check the syntax of the specifications and at the same time gives a flavour of step by step execution of the specifications.

By the time when we have used XTL, it was an ongoing research project and we did not carry any extensive experiment with it by using cCSP. Although some experiments have been carried out in XTL by using StAC, which are shown in [8], where some case studies were modelled in XTL and some simple CTL [33, 34] properties were verified by using the tool, we have used the tool only as an animator for the transition rules of cCSP. The encoding of the transition rules was simple. The animation is the representation of the encoded rules. The animation shows whether the animated transitions are according to our expectation. If there is any transition that is not as expected then it is either because of wrong encoding of the transition rules or there is an error in the transition rule. In our simple experiments, we have found both types of errors where there were mistakes in the transition rules as well as there were errors in the encoding. We find that it is a very useful tool to help to understand the operational semantics and can be used as a supporting tool to teaching cCSP. The encodings of all the operators in XTL are available in Appendix A.

3.4   Related Work

The semantics of StAC is given by defining the operational semantics of a variant of StAC, called StAC_i (StAC with indexes). Several simultaneous compensations can be associated with a process in StAC_i. A process decides which task to attach to the compensation activities. Indexes are used to distinguish different compensation tasks. The operators that deal with compensations are indexed by the compensation task index to which they apply. The compensation information of a process is maintained by a compensation function that maps each compensation task index to a compensation process. The operational semantics is defined by using transitions between configurations. A configuration is defined as a tuple:

$$(P,\ C,\ \sigma)\ \in\ Process\ \times\ (I \rightarrow\ Process) \times \Sigma$$

where C is a compensation function for the process P that returns the compensation process $C(i)$ for each compensation index i. Data states are included in StAC and represented by Σ. A labelled transition is defined as follows:

$$(P,\ C,\ \sigma)\ \xrightarrow{A}\ (P',\ C',\ \sigma')$$

The above transition denotes that activity A may cause a configuration transition from $(P,\ C,\ \sigma)$ to $(P',\ C',\ \sigma')$. Transition rules are defined for the primitives: *Reverse* and *Accept*, where the reverse operator causes the compensation task to be executed and the accept operator clears the compensation task. Both operators can be indexed. The semantics of compensation is defined by adding an index to each compensation operator in a term corresponding to the scope in which it appears. Unlike cCSP, transition rules are defined for silent transitions, labelled by τ, where the operation is not visible to the external environment. Consider the following rule for a compensation pair, where the compensation Q is added to the compensation function C when the primary process of the pair terminates successfully:

$$\overline{(skip \div_i Q,\ C,\ \sigma)\ \xrightarrow{\tau}\ (skip, C[i := (Q\ ;\ C(i))],\ \sigma)}$$

Here the compensation task i is set to Q in sequence with the previous compensation, denoted by $C[i := (Q\ ;\ C(i))]$.

Comparing to cCSP, StAC provides a large number of operators. The operational semantics are defined by using the indexed version StAC_i, which is somewhat complicated. cCSP provides a neater operational semantics definition than StAC.

Bruni *et al* [21] have developed an operational semantics for a language with similar operators to cCSP, including compensation pairs and transaction blocks (or sagas as they call them). As in cCSP, and unlike StAC, the invocation of compensation in a

saga is automatic depending on failure or success, which leads to a neater operational semantics. However, unlike cCSP, the operational semantics in [21] is defined by using big-step semantics. Big-step semantics describe how the overall results of the execution are obtained. The big step semantics are closer to the trace semantics while our small-step semantics describes how compensating processes should be executed. A comparison of the operators of cCSP and the language described in [21] may be found in [19].

3.5 Conclusion

This chapter illustrates the operational semantics of cCSP by using labelled transition systems. Semantics are given for both standard as well as compensable processes. One of the advantages of the availability of an operational semantics of a language is the increasing understanding and the possibility of formal reasoning that this brings. Operational semantics describes the individual steps of the computation that take place and it is very close to the actual implementation. A Prolog implementation of the semantics is also defined in XTL showing animation of the semantics. It is our future plan to develop tool support for animating and model checking cCSP specifications. Details are placed in Future Work section Section 8.2.

Chapter 4

Relating Semantic Models

4.1 Introduction

Operational and denotational semantics are two well-known methods of assigning meaning to programming languages and both semantics play a significant role for a full description of the language. Since both semantics have valuable applications, we want to use them both. The key question is- *"how they are related"*?

Different semantic models are not in competition, but are mutually complementary and serve different purposes. Regarding defining semantic models, Mosses [82] advocated that,

a) Two or more complementary semantic descriptions should be provided for each language, in different styles and

b) Prove that the semantic descriptions are equivalent.

Having defined both operational and denotational semantic models for cCSP [27, 29], we have already fulfilled the first requirement. In this chapter, we are interested in the second point to define and prove a relationship between the semantic models.

4.1.1 Chapter Contribution

This chapter draws the correspondence of two different semantic representations of compensating CSP. We define and prove a formal relationship between the two semantic models of cCSP. The correspondence is derived by applying a systematic approach. Traces are extracted from the transition rules and by structural induction over the process terms, it is shown that the derived traces correspond to the originally defined traces. The correspondence proofs are carried out for both standard and compensable processes.

Showing the correspondence establishes a strong foundation for the language. Proving the correspondence means that any of the presentation can be accepted as a primary definition of the meaning of the language and each definition can be used correctly at different times, and for different purposes.

4.1.2 Chapter Structure

The remainder of the chapter is organised as follows. In Section 4.2, first we describe how to derive traces from the operational rules and then we define the theorems that show the correspondence between the two semantic models. In the following subsections, we define and prove lemmas for each operator that support the proof of the theorems. For each operator, lemmas are defined for both standard and compensable processes. In Section 4.3 we compare our work with some similar research work. Finally, in Section 4.4 we draw the conclusion of the chapter.

4.2 Semantic Correspondence

In this section, we show how we define and prove a formal relationship between the two semantic models of cCSP. We have adopted a systematic approach to derive the correspondence between the semantic models. The correspondence is derived in two steps:

– Traces are extracted from the operational rules, and

– The correspondence is derived between the extracted traces with the original semantic definition.

The steps that we follow are depicted in Figure 4.1.

The operational semantics leads to lifted transition relations labelled by sequences of events. This is defined recursively. For a standard process P:

$$
\begin{aligned}
P \xrightarrow{\langle \omega \rangle} Q &= P \xrightarrow{\omega} Q \\
P \xrightarrow{\langle a \rangle t} Q &= \exists P' \cdot P \xrightarrow{a} P' \wedge P' \xrightarrow{t} Q
\end{aligned}
$$

Then for a standard process P, we define the derived trace $DT(P)$ as follows:

Definition 4.1 (Derived trace: standard process). For a trace t

$$
t \in DT(P) = P \xrightarrow{t} 0
$$
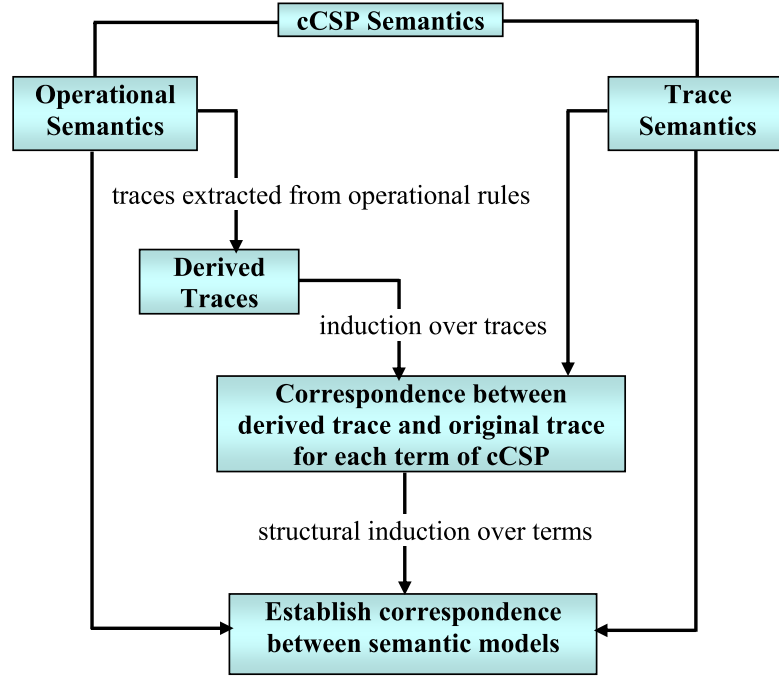
FIGURE 4.1: Steps to establish relationship between semantic models

The trace t consists of a sequence of events (Σ^*), followed by a terminal event ω where, $\omega \in \{\checkmark, !, ?\}$.

Compensable processes have both forward and compensation behaviour. A compensable process is defined as a pair of traces of the form: $(p\langle\omega\rangle, p'\langle\omega'\rangle)$, where $p\langle\omega\rangle$ is the forward behaviour and $p'\langle\omega'\rangle$ is the compensation. Hence, it is required to extract traces from both forward and compensation behaviour. The forward behaviour of a compensable process PP is defined as follows:

$$PP \xrightarrow{t} R \quad (t \text{ ends with } \omega)$$

where t is the trace of the forward behaviour. Note that the trace t ends with a terminal event ω. R is the attached compensation. The behaviour of compensation is similar to standard processes and by reusing that we get the following definition:

$$PP \xrightarrow{(t,t')} 0 \quad = \quad \exists R \cdot PP \xrightarrow{t} R \wedge R \xrightarrow{t'} 0$$

where t' is the trace of the compensation. By using this definition we get the trace derivation rule for a compensable process PP, which is defined as follows:

Definition 4.2 (Derived trace: compensable process). For traces t and t'

$$(t, t') \in DT(PP) \quad = \quad PP \xrightarrow{(t,t')} 0$$

Consider the trace semantics of P and PP, defined in Section 2.6.1 as $T(P)$ and $T(PP)$ respectively. The correspondence between the semantic models is then defined by the following theorem:

Theorem 4.3. *For any standard process term P, where $P \neq 0$*

$$DT(P) \;=\; T(P)$$

For any compensable process terms PP, where $PP \neq 0$ and does not contain the term $\langle PP, P \rangle$,

$$DT(PP) \;=\; T(PP)$$

The auxiliary construct $\langle PP, P \rangle$ was introduced in Chapter 3 in order to define the operational semantics for compensable sequential composition. Theorem 4.3 is proved by using mutual structural induction. We outline the structural cases for the proof of the theorem in the following sections.

In order to derive the correspondence, traces are extracted for each term of the language, and its correspondence is shown with the corresponding traces in the trace semantics. Assume P and Q are standard process terms, then for all the operators, we prove that

$$t \in DT(P \otimes Q) \;=\; t \in T(P \otimes Q) \tag{4.1}$$

For each such operator $\otimes$, the proof is performed by induction over traces. In the proof we assume that,

$$DT(P) \;=\; T(P)$$
$$DT(Q) \;=\; T(Q)$$

Like standard processes, we derive traces for all the operators for compensable processes and show that

$$(t, t') \in DT(PP \otimes QQ) \;=\; (t, t') \in T(PP \otimes QQ) \tag{4.2}$$

In order to carry on proving the correspondence, we prove the following theorem concerning the evolution of a standard process term to a null process.

Theorem 4.4. *For any process term P, where $P \neq 0$, there exists a trace t of P. That is:*

$$\forall P \cdot P \neq 0 \;\Rightarrow\; \exists t \cdot P \xrightarrow{\ t\ } 0$$

A similar theorem is also defined for compensable process terms.

Theorem 4.5. *For any process term PP, where $PP \neq 0$, there exists a forward trace t of PP. That is:*

$$\forall PP \cdot PP \neq 0 \;\Rightarrow\; \exists P, t \cdot PP \xrightarrow{\;t\;} P$$

Both theorems 4.4 and 4.5 can be proved by using structural induction over process terms.

The correspondence proofs for the process terms are shown in the following sections.

4.2.1 Sequential Composition

In this section, we demonstrate the correspondence between the semantic models for sequential composition of both standard and compensable processes.

Standard Processes

For standard processes P and Q, the correspondence between the semantic models are derived by showing that

$$t \in DT(P \; ; Q) \quad = \quad t \in T(P \; ; Q)$$

By Definition 4.1, the derived traces for the sequential composition are as follows:

$$t \in DT(P \; ; Q) \quad = \quad (P \; ; Q) \xrightarrow{\;t\;} 0$$

We also expand the definition of trace semantic of the sequential composition as follows:

$$
\begin{aligned}
t \in\; & T(P \; ; Q) \\
= &\; \exists p, q \cdot t = (p \; ; q) \;\wedge\; p \in T(P) \;\wedge\; q \in T(Q) && \text{[Trace definition]} \\
= &\; \exists p, q \cdot t = (p \; ; q) \;\wedge\; p \in DT(P) \;\wedge\; q \in DT(Q) && \text{[Induction assumption]} \\
= &\; \exists p, q \cdot t = (p \; ; q) \;\wedge\; P \xrightarrow{\;p\;} 0 \;\wedge\; Q \xrightarrow{\;q\;} 0 && \text{[Derived trace definition]}
\end{aligned}
$$

Finally, from the above definitions of traces, the following lemma is derived to prove the correspondence between them:

Lemma 4.6. $(P \; ; Q) \xrightarrow{\;t\;} 0 \;=\; \exists p, q \cdot t = (p \; ; q) \;\wedge\; P \xrightarrow{\;p\;} 0 \;\wedge\; Q \xrightarrow{\;q\;} 0$

The lemma is proved by using induction over traces. The induction is based on the following two cases:

– *basic step*: the lemma is proved for trace $\langle \omega \rangle$.

– *inductive step*: the lemma is proved for trace $a\langle t \rangle$ assuming it true for trace t.

In order to support the inductive proof of the lemma, we have derived the following equations from the transition rules of sequential composition. The equations are derived based on the events by which the transition rules are defined. From rules 3.5 and 3.6 in page 32, we derive the following equation:

$$(P \ ; \ Q) \xrightarrow{\omega} 0 \ = \ P \xrightarrow{\checkmark} 0 \wedge Q \xrightarrow{\omega} 0$$
$$\vee \ P \xrightarrow{\omega} 0 \wedge \omega \neq \checkmark \tag{4.3}$$

Similarly the following equation is derived from the rules 3.4 and 3.5 in page 32:

$$(P \ ; \ Q) \xrightarrow{a} R \ = \ \exists P' \cdot P \xrightarrow{a} P' \wedge R = (P' \ ; \ Q)$$
$$\vee \ P \xrightarrow{\checkmark} 0 \wedge Q \xrightarrow{a} R \tag{4.4}$$

Equation 4.3 is used in the base case of the inductive proof and equation 4.4 is used in inductive case of the proof.

Proof:
Basic Step: Case - $\langle \omega \rangle$

$$(P \ ; \ Q) \xrightarrow{\langle \omega \rangle} 0$$
$$= \ (P \ ; \ Q) \xrightarrow{\omega} 0$$
$$= \ \text{"From derived Equation 4.3"}$$

$$P \xrightarrow{\checkmark} 0 \wedge Q \xrightarrow{\omega} 0 \tag{4.5}$$
$$\vee \ P \xrightarrow{\omega} 0 \wedge \omega \neq \checkmark \tag{4.6}$$

From Equation 4.5

$$P \xrightarrow{\checkmark} 0 \wedge Q \xrightarrow{\omega} 0$$
$$= \ \exists p, q \cdot p = \langle \checkmark \rangle \wedge q = \langle \omega \rangle \wedge P \xrightarrow{p} 0 \wedge Q \xrightarrow{q} 0$$
$$= \ \text{"By trace rule: } (p \ ; \ q) = (\langle \checkmark \rangle \ ; \ \langle \omega \rangle) = \langle \omega \rangle \text{"}$$
$$\exists p, q \cdot \langle \omega \rangle = (p \ ; \ q) \wedge p = \langle \checkmark \rangle \wedge P \xrightarrow{p} 0 \wedge Q \xrightarrow{q} 0$$

From Equation 4.6

$$P \xrightarrow{\omega} 0 \ \wedge \ \omega \neq \checkmark$$

$=$ "By using Theorem 4.4"

$$\exists \, p \cdot p = \langle \omega \rangle \ \wedge \ \omega \neq \checkmark \ \wedge \ P \xrightarrow{p} 0 \ \wedge \ \exists \, q \cdot Q \xrightarrow{q} 0$$

$=$ "By using trace rule: $\langle \omega \rangle \ ; \ q \ = \ \langle \omega \rangle$ where $\ \omega \neq \checkmark$"

$$\exists \, p, q \cdot p = \langle \omega \rangle \ \wedge \ \omega \neq \checkmark \ \wedge \ \langle \omega \rangle = (p \ ; \ q) \ \wedge \ P \xrightarrow{p} 0 \ \wedge \ Q \xrightarrow{q} 0$$

$= \ \exists \, p, q \cdot \langle \omega \rangle = (p \ ; \ q) \ \wedge \ p \neq \langle \checkmark \rangle \ \wedge \ P \xrightarrow{p} 0 \ \wedge \ Q \xrightarrow{q} 0$

Therefore, for $\ \langle \omega \rangle$, combining the above two derivations:

$$\exists \, p, q \cdot \langle \omega \rangle = (p \ ; \ q) \ \wedge \ p = \langle \checkmark \rangle \ \wedge \ P \xrightarrow{p} 0 \ \wedge \ Q \xrightarrow{q} 0$$

$$\vee \ \exists \, p, q \cdot \langle \omega \rangle = (p \ ; \ q) \ \wedge \ p \neq \langle \checkmark \rangle \ \wedge \ P \xrightarrow{p} 0 \ \wedge \ Q \xrightarrow{q} 0$$

$$= \ \exists \, p, q \cdot \langle \omega \rangle = (p \ ; \ q) \ \wedge \ P \xrightarrow{p} 0 \ \wedge \ Q \xrightarrow{q} 0$$

Inductive Step: Case - $\ \langle a \rangle t$

$$(P \ ; \ Q) \xrightarrow{\langle a \rangle t} 0$$

$= \ \exists \, R \cdot (P \ ; \ Q) \xrightarrow{a} R \ \wedge \ R \xrightarrow{t} 0$

$=$ "From derived Equation 4.4"

$$\exists \, P' \cdot P \xrightarrow{a} P' \ \wedge \ (P' \ ; \ Q) \xrightarrow{t} 0 \qquad\qquad (4.7)$$

$$\vee \ \exists \, R \cdot P \xrightarrow{\checkmark} 0 \ \wedge \ Q \xrightarrow{a} R \ \wedge \ R \xrightarrow{t} 0 \qquad\qquad (4.8)$$

From Equation 4.7

$$\exists \, P' \cdot P \xrightarrow{a} P' \ \wedge \ (P' \ ; \ Q) \xrightarrow{t} 0$$

$=$ "Inductive hypothesis"

$$\exists \, P' \cdot P \xrightarrow{a} P' \ \wedge \ \exists \, p', q \cdot t = (p' \ ; \ q) \ \wedge \ P' \xrightarrow{p'} 0 \ \wedge \ Q \xrightarrow{q} 0$$

$=$ "Removing P' "

$$\exists \, p', q \cdot t = (p' \ ; \ q) \ \wedge \ P \xrightarrow{\langle a \rangle p'} 0 \ \wedge \ Q \xrightarrow{q} 0$$

$=$ "Using trace rule $\ \langle a \rangle t \ = \ \langle a \rangle (p' \ ; \ q) \ = \ (\langle a \rangle p') \ ; \ q$"

$$\exists \, p', q \cdot \langle a \rangle t = (\langle a \rangle p' \ ; \ q) \ \wedge \ P \xrightarrow{\langle a \rangle p'} 0 \ \wedge \ Q \xrightarrow{q} 0$$

$$= \ \exists \, p, p', q \cdot p = \langle a \rangle p' \ \wedge \ \langle a \rangle t = (p \ ; \ q) \ \wedge \ P \xrightarrow{p} 0 \ \wedge \ Q \xrightarrow{q} 0$$

From Equation 4.8

$$\exists R \cdot P \xrightarrow{\checkmark} 0 \ \wedge \ Q \xrightarrow{a} R \ \wedge \ R \xrightarrow{t} 0$$

$$= \ P \xrightarrow{\checkmark} 0 \ \wedge \ Q \xrightarrow{\langle a \rangle t} 0$$

$$= \ \exists p, q \cdot p = \langle \checkmark \rangle \ \wedge \ q = \langle a \rangle t \ \wedge \ P \xrightarrow{p} 0 \ \wedge \ Q \xrightarrow{q} 0$$

$$= \ \text{``By trace rule: } \langle \checkmark \rangle \ ; \ q = q\text{''}$$

$$\exists p, q \cdot \langle a \rangle t = (p \ ; \ q) \ \wedge \ p = \langle \checkmark \rangle \ \wedge \ P \xrightarrow{p} 0 \ \wedge \ Q \xrightarrow{q} 0$$

Therefore, for $\langle a \rangle t$, from (4.7) $\vee$ (4.8)

$$\exists p, p', q \cdot p = \langle a \rangle p' \ \wedge \ \langle a \rangle t = (p \ ; \ q) \ \wedge \ P \xrightarrow{p} 0 \ \wedge \ Q \xrightarrow{q} 0$$

$$\vee \ \exists p, q \cdot p = \langle \checkmark \rangle \ \wedge \ \langle a \rangle t = (p \ ; \ q) \ \wedge \ P \xrightarrow{p} 0 \ \wedge \ Q \xrightarrow{q} 0$$

$$= \ \text{``Combining existential quantifications''}$$

$$\exists p, p', q \cdot (p = \langle \checkmark \rangle \ \vee \ p = \langle a \rangle p') \ \wedge \ \langle a \rangle t = (p \ ; \ q) \ \wedge \ P \xrightarrow{p} 0 \ \wedge \ Q \xrightarrow{q} 0$$

$$= \ \text{``} p \ ; \ q = \langle a \rangle t \ \Rightarrow \ p \neq \langle ! \rangle \ \wedge \ p \neq \langle ? \rangle \text{''}$$

$$\exists p, q \cdot \langle a \rangle t = (p \ ; \ q) \ \wedge \ P \xrightarrow{p} 0 \ \wedge \ Q \xrightarrow{q} 0 \qquad \qquad \square$$

Compensable Processes

For sequential composition of compensable processes PP and QQ, the correspondence between the semantic models is given by,

$$(t, t') \in DT(PP \ ; \ QQ) \quad = \quad (t, t') \in T(PP \ ; \ QQ) \qquad (4.9)$$

For a compensable process, it is required to derive traces from both forward and compensation behaviour. The behaviour of compensation is similar to that of standard process and the proofs from standard processes can be reused. Let $(t, t') \in DT(PP \ ; \ QQ)$. According to the definition of derived traces we can define,

$$
\begin{aligned}
(t, t') \ &\in \ DT(PP \ ; \ QQ) \\
&= \ (PP \ ; \ QQ) \xrightarrow{(t, t')} 0 \\
&= \ \exists R \cdot (PP \ ; \ QQ) \xrightarrow{t} R \ \wedge \ R \xrightarrow{t'} 0
\end{aligned}
$$

For compensable processes, we only define the lifted forward behaviour and reuse the proof from the standard processes for the compensations. The sequential composition of compensable processes is defined in such a way that when the first process terminates successfully, the next process starts to execute and after its termination the compensations from both processes are accumulated in reverse order. However, if the first process terminates unsuccessfully, then the next process will not execute and only the compen-

sations of the first process will be stored. The following lemma is defined to derive the lifted forward traces from compensable sequential composition:

Lemma 4.7. $(PP \mathbin{;} QQ) \xrightarrow{t} R = \exists P, Q, p, q \cdot t = (p \mathbin{;} q) \wedge PP \xrightarrow{p} P \wedge$
$$QQ \xrightarrow{q} Q \wedge R = COND\,(last(p) = \checkmark, (Q \mathbin{;} P), P)$$

Here $COND$ is a conditional expression and it is defined as follows:

$$COND(true, e1, e2) = e1$$
$$COND(false, e1, e2) = e2$$

We use an operator on traces: $last(p)$, which returns the terminal event of the trace p. The lemma considers the cases where the first process PP terminates with or without $\langle \checkmark \rangle$. The conditional expression $COND$ evaluates the accumulated compensations. When the first process of the composition terminates successfully $(last(p) = \checkmark)$ $COND$ returns the accumulated compensation from both the processes (P, Q). In the other case where the first process terminates with an interrupt (by throw or yield), $COND$ returns only the compensation from the first process (P) as in this case the second process QQ will not execute.

In order to support the inductive proof of Lemma 4.7, equations are derived from operational rules. The following equation is derived from rules 3.15 and 3.17 in page 34: derive

$$(PP \mathbin{;} QQ) \xrightarrow{\omega} R = PP \xrightarrow{\checkmark} P \wedge QQ \xrightarrow{\omega} Q \wedge R = (Q \mathbin{;} P)$$
$$\vee \quad PP \xrightarrow{\omega} P \wedge \omega \neq \checkmark \wedge R = P \qquad (4.10)$$

The following equation is derived from rules 3.14 and 3.16:

$$(PP \mathbin{;} QQ) \xrightarrow{a} RR = PP \xrightarrow{a} PP' \wedge RR = (PP' \mathbin{;} QQ)$$
$$\vee \quad PP \xrightarrow{\checkmark} P \wedge QQ \xrightarrow{a} QQ' \wedge RR = \langle QQ', P \rangle \quad (4.11)$$

Supported by the above two equations, we prove the lemma by induction. Detailed proof steps are placed in Appendix B. However, in the inductive step of the proof, the auxiliary construct $\langle QQ, P \rangle$ is involved:

$$PP \mathbin{;} QQ \xrightarrow{\langle a \rangle t} R = \exists RR \cdot PP \mathbin{;} QQ \xrightarrow{a} RR \wedge RR \xrightarrow{t} R$$
$$= \text{``From equation 4.11''}$$
$$\exists PP' \cdot PP \xrightarrow{a} PP' \wedge PP' \mathbin{;} QQ \xrightarrow{t} R$$
$$\vee \exists P, QQ' \cdot PP \xrightarrow{\checkmark} P \wedge QQ \xrightarrow{a} QQ' \wedge \langle QQ', P \rangle \xrightarrow{t} R$$
$$(4.12)$$

To deal with this, we need another lemma which will support the removal of the auxiliary construct in equation 4.12. We know that the auxiliary construct is introduced in the situation where the forward behaviour of the first process of the sequential composition is terminated with $\checkmark$ and its compensation is stored and the second process of the composition has started. The lemma is defined as follows:

Lemma 4.8. $\quad \langle QQ, P \rangle \xrightarrow{t} R = \exists Q \cdot QQ \xrightarrow{t} Q \wedge R = (Q \; ; P)$

Similarly to other lemmas, this lemma is also proved by induction over the traces.
Proof:
Basic step: $t = \langle \omega \rangle$

$$\langle QQ, P \rangle \xrightarrow{\langle \omega \rangle} R$$
$$= \langle QQ, P \rangle \xrightarrow{\omega} R$$
$$= \text{``From operational rule 3.19 ''}$$
$$\exists Q \cdot QQ \xrightarrow{\omega} Q \wedge R = (Q \; ; P)$$

Inductive step: $t = \langle a \rangle t$

$$\langle QQ, P \rangle \xrightarrow{\langle a \rangle t} R$$
$$= \exists RR \cdot \langle QQ, P \rangle \xrightarrow{a} RR \wedge RR \xrightarrow{t} R$$
$$= \text{``From operational rule 3.18 ''}$$
$$\exists QQ' \cdot QQ \xrightarrow{a} QQ' \wedge \langle QQ', P \rangle \xrightarrow{t} R$$
$$= \text{``Inductive hypothesis''}$$
$$\exists QQ' \cdot QQ \xrightarrow{a} QQ' \wedge \exists Q \cdot QQ' \xrightarrow{t} Q \wedge R = (Q \; ; P)$$
$$= \text{``Removing } QQ' \text{ ''}$$
$$\exists Q \cdot QQ \xrightarrow{\langle a \rangle t} Q \wedge R = (Q \; ; P) \qquad \qquad \square$$

Now it is possible to continue the proof of the lemma 4.7. Completing the proof derives the traces from the forward behaviour of the composition. By reusing the proofs from the standard processes for compensation, now prove the initially defined equation 4.9, which is:

$$(t, t') \in DT(PP \; ; QQ) = (t, t') \in T(PP \; ; QQ)$$

Proof of Equation 4.9:

$$(t, t') \quad \in DT(PP \; ; \; QQ)$$

$$= (PP \; ; \; QQ) \xrightarrow{(t,t')} 0$$

$$= \exists R \cdot (PP \; ; \; QQ) \xrightarrow{t} R \; \wedge \; R \xrightarrow{t'} 0$$

$$= \text{``By using Lemma 4.7''}$$

$$\exists P, Q, R, p, q \cdot t = (p \; ; \; q) \; \wedge \; PP \xrightarrow{p} P \; \wedge \; QQ \xrightarrow{q} Q$$

$$\wedge \; R = COND(last(p) = \checkmark, (Q \; ; \; P), P) \; \wedge \; R \xrightarrow{t'} 0$$

We now consider both the cases where p ends with and without $\checkmark$ and we separate these two conditions. The sequential operator is defined in such a way that when $last(p) \neq \checkmark$, the traces of QQ are discarded. We continue the proof where p is replaced by $p\langle\checkmark\rangle$ $(R = Q \; ; \; P)$ and by $p\langle\omega\rangle$ $(R = P)$.

$$= \exists P, Q, p, q \cdot t = (p\langle\checkmark\rangle \; ; \; q) \; \wedge \; PP \xrightarrow{p\langle\checkmark\rangle} P \; \wedge \; QQ \xrightarrow{q} Q \; \wedge \; (Q \; ; \; P) \xrightarrow{t'} 0$$

$$\vee \; \exists P, p \cdot t = p\langle\omega\rangle \; \wedge \; \omega \neq \checkmark \; \wedge \; PP \xrightarrow{t} P \; \wedge \; P \xrightarrow{t'} 0$$

$$= \text{``By using Lemma 4.6''}$$

$$\exists P, Q, p, q \cdot (t = (p\langle\checkmark\rangle \; ; \; q) \; \wedge \; PP \xrightarrow{p\langle\checkmark\rangle} P \; \wedge \; QQ \xrightarrow{q} Q$$

$$\wedge \; \exists p', q' \cdot t' = (q' \; ; \; p') \; \wedge \; Q \xrightarrow{q'} 0 \; \wedge \; P \xrightarrow{p'} 0)$$

$$\vee \; \exists P, p, p' \cdot t = p\langle\omega\rangle \; \wedge \; \omega \neq \checkmark \; \wedge \; t' = p' \; \wedge \; PP \xrightarrow{p\langle\omega\rangle} P \; \wedge \; P \xrightarrow{p'} 0$$

$$= \text{``Removing } P, Q\text{''}$$

$$\exists p, p', q, q' \cdot t = (p\langle\checkmark\rangle \; ; \; q) \; \wedge \; t' = (q' \; ; \; p') \; \wedge \; PP \xrightarrow{p\langle\checkmark\rangle, p'} 0 \; \wedge \; QQ \xrightarrow{q, q'} 0$$

$$\vee \; \exists p, p' \cdot t = p\langle\omega\rangle \; \wedge \; t' = p' \; \wedge \; PP \xrightarrow{p\langle\omega\rangle, p'} 0 \; \wedge \; \omega \neq \checkmark$$

$$= \text{``By definition of derived traces''}$$

$$\exists p, p', q, q' \cdot t = (p\langle\checkmark\rangle \; ; \; q) \; \wedge \; t' = (q' \; ; \; p')$$

$$\wedge \; (p\langle\checkmark\rangle, p') \in DT(PP) \; \wedge \; (q, q') \in DT(QQ)$$

$$\vee \; \exists p, p' \cdot t = p\langle\omega\rangle \; \wedge \; \omega \neq \checkmark \; \wedge \; t' = p' \; \wedge \; (p\langle\omega\rangle, p') \in DT(PP)$$

$$= \text{``Structural induction assumption''}$$

$$\exists p, p', q, q' \cdot t = (p\langle\checkmark\rangle \; ; \; q) \; \wedge \; t' = (q' \; ; \; p')$$

$$\wedge \; (p\langle\checkmark\rangle, p') \in T(PP) \; \wedge \; (q, q') \in T(QQ)$$

$$\vee \; \exists p, p' \cdot t = p\langle\omega\rangle \; \wedge \; \omega \neq \checkmark \; \wedge \; t' = p' \; \wedge (p\langle\omega\rangle, p') \in T(PP)$$

We have $(t, t') = ((p\langle\checkmark\rangle \; ; \; q), (q' \; ; \; p'))$ or $(t, t') = (p\langle\omega\rangle, p')$
Using trace rules we can write that:
$(t, t') = (p\langle\checkmark\rangle \; ; \; q), (q' \; ; \; p') = (p\langle\checkmark\rangle, p') \; ; \; (q, q')$
Similarly,

$(t, t') \; = \; (p\langle\omega\rangle, p') = (p\langle\omega\rangle, p') \; ; \; (q, q') \; \mathbf{where} \; \omega \neq \checkmark$

$=$ "By using the definition of sequential composition over traces"

$\quad \exists\, p, p', q, q' \cdot (t, t') = (p, p') \; ; \; (q, q') \; \wedge \; (p, p') \in T(PP) \; \wedge \; (q, q') \in T(QQ)$

$=$ "By trace rule"

$\quad (t, t') \in T(PP \; ; \; QQ)$ $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ $\square$

4.2.2   Interrupt Handler

For standard processes P and Q, and following the equation 4.1, for the interrupt handler operator we prove that,

$$t \; \in \; DT(P \triangleright Q) \;\; = \;\; t \; \in \; T(P \triangleright Q) \qquad\qquad (4.13)$$

In order to prove the above equation, we extend the definition of both traces and derived traces and define the following lemma:

Lemma 4.9. $(P \triangleright Q) \xrightarrow{t} 0 \; = \; \exists\, p, q \cdot t = (p \triangleright q) \wedge P \xrightarrow{p} 0 \wedge Q \xrightarrow{q} 0$

We derive the following supporting equations in order to support the inductive proof of the above lemma:

$$
\begin{aligned}
(P \triangleright Q) \xrightarrow{\omega} 0 \; &= \; P \xrightarrow{!} 0 \wedge Q \xrightarrow{\omega} 0 \\
&\quad \vee \; P \xrightarrow{\omega} 0 \wedge \omega \neq \,! \qquad\qquad (4.14)
\end{aligned}
$$

$$
\begin{aligned}
(P \triangleright Q) \xrightarrow{a} R \; &= \; \exists\, P' \cdot P \xrightarrow{a} P' \wedge R = (P' \triangleright Q) \\
&\quad \vee \; P \xrightarrow{!} 0 \wedge Q \xrightarrow{a} R \qquad\qquad (4.15)
\end{aligned}
$$

We know that the interrupt handler operator is dual to the sequential composition operator. In sequential composition, control passes from one process to another by a $\checkmark$, whereas in interrupt handler, control passes from one process to another by an interrupt !. Hence, the inductive proof steps of the above lemma is similar to that of standard sequential composition. The proof steps are omitted from here and placed in the Appendix B.

4.2.3   Parallel Composition

The parallel composition of two traces is defined as the interleaving of the observable events of the traces, followed by the synchronisation of their terminal events. For example, consider A and B are asynchronous events. The the execution of $A \parallel B$ is either A followed by B or vice versa. For traces p and q,(not including terminal events) we write

$p \mid\mid\mid q$ to denote the set of interleaving of p and q. The interleaving of these two traces has the following definition:

$$
\begin{aligned}
\langle\rangle \in (p \mid\mid\mid q) \quad &= \quad p = \langle\rangle \;\wedge\; q = \langle\rangle \\
\langle a\rangle t \in (p \mid\mid\mid q) \quad &= \quad \exists\, p' \cdot p = \langle a\rangle p' \;\wedge\; t \in (p' \mid\mid\mid q) \\
&\vee \quad \exists\, q' \cdot q = \langle a\rangle q' \;\wedge\; t \in (p \mid\mid\mid q')
\end{aligned}
$$

Standard Processes

Following equation 4.1, for standard processes P and Q, in this section we prove that,

$$
t \in DT(P \parallel Q) \quad = \quad t \in T(P \parallel Q) \tag{4.16}
$$

The equation is proved by using structural induction. In order to prove the equation, we follow the same approach as for sequential composition and extend the definitions of trace as well as derived traces and define the following lemma:

Lemma 4.10. $\quad (P \parallel Q) \xrightarrow{\;t\;} 0 \;=\; \exists\, p, q \cdot t \in (p \parallel q) \;\wedge\; P \xrightarrow{\;p\;} 0 \;\wedge\; Q \xrightarrow{\;q\;} 0$

The lemma is proved by applying induction over traces, where traces $\langle\omega\rangle$ and $\langle a\rangle t$ are considered as basic and inductive cases respectively. To support the inductive proof of the lemma, we derive the following two equations from the transition rules 3.10 and 3.11 (page 33) of the operational semantics of parallel composition. As mentioned earlier, the equations are categorised according to the events by which the transition rules are defined.

$$
\begin{aligned}
P \parallel Q \xrightarrow{\;\omega\;} 0 \quad &= \quad P \xrightarrow{\;\omega 1\;} 0 \;\wedge\; Q \xrightarrow{\;\omega 2\;} 0 \;\wedge\; \omega = \omega 1 \,\&\, \omega 2 \tag{4.17} \\
P \parallel Q \xrightarrow{\;a\;} R \quad &= \quad \exists\, P' \cdot P \xrightarrow{\;a\;} P' \;\wedge\; R = (P' \parallel Q) \\
&\vee \quad \exists\, Q' \cdot Q \xrightarrow{\;a\;} Q' \;\wedge\; R = (P \parallel Q') \tag{4.18}
\end{aligned}
$$

By using the above two equations, one for the base case and the other for the inductive case, we can prove the lemma. The proof of the basic step is simple and placed in Appendix B. The inductive step of the proof is shown here.

Inductive step: Case - $\langle a \rangle t$

$$(P \parallel Q) \xrightarrow{\langle a \rangle t} 0$$

$= \exists R \cdot (P \parallel Q) \xrightarrow{\langle a \rangle} R \wedge R \xrightarrow{t} 0$

$= \exists R \cdot (P \parallel Q) \xrightarrow{a} R \wedge R \xrightarrow{t} 0$

$=$ "By using derived equation 4.18"

$\qquad \exists P' \cdot P \xrightarrow{a} P' \wedge (P' \parallel Q) \xrightarrow{t} 0$

$\vee \ \exists Q' \cdot Q \xrightarrow{a} Q' \wedge (P \parallel Q') \xrightarrow{t} 0$

$=$ "Inductive hypothesis"

$\qquad \exists P' \cdot P \xrightarrow{a} P' \wedge \exists p', q \cdot t \in (p' \parallel q) \wedge P' \xrightarrow{p'} 0 \wedge Q \xrightarrow{q} 0$

$\vee \ \exists Q' \cdot Q \xrightarrow{a} Q' \wedge \exists p, q' \cdot t \in (p \parallel q') \wedge P \xrightarrow{p} 0 \wedge Q' \xrightarrow{q'} 0$

$=$ "Removing P' and Q'"

$\qquad \exists p', q \cdot t \in (p' \parallel q) \wedge P \xrightarrow{\langle a \rangle p'} 0 \wedge Q \xrightarrow{q} 0$

$\vee \ \exists p, q' \cdot t \in (p \parallel q') \wedge P \xrightarrow{p} 0 \wedge Q \xrightarrow{\langle a \rangle q'} 0$

$= \exists p, p', q \cdot p = \langle a \rangle p' \wedge t \in (p' \parallel q) \wedge P \xrightarrow{p} 0 \wedge Q \xrightarrow{q} 0$

$\vee \ \exists p, q, q' \cdot q = \langle a \rangle q' \wedge t \in (p \parallel q') \wedge P \xrightarrow{p} 0 \wedge Q \xrightarrow{q} 0$

$=$ "By trace rule: $t \in (p' \parallel q) \Rightarrow \langle a \rangle t \in (\langle a \rangle p' \parallel q)$

$\qquad$ Similarly, $t \in (p \parallel q') \Rightarrow \langle a \rangle t \in (\langle a \rangle p \parallel q')$"

$\qquad \exists p, p', q \cdot p = \langle a \rangle p' \wedge \langle a \rangle t \in (\langle a \rangle p' \parallel q) \wedge P \xrightarrow{p} 0 \wedge Q \xrightarrow{q} 0$

$\vee \ \exists p, q, q' \cdot q = \langle a \rangle q' \wedge \langle a \rangle t \in (p \parallel \langle a \rangle q') \wedge P \xrightarrow{p} 0 \wedge Q \xrightarrow{q} 0$

$=$ "By the definition of traces"

$\qquad \exists p, q \cdot \langle a \rangle t \in (p \parallel q) \wedge P \xrightarrow{p} 0 \wedge Q \xrightarrow{q} 0$

Compensable Process

For compensable processes PP and QQ, we prove the following equation for parallel composition:

$$(t, t') \in DT(PP \parallel QQ) \quad = \quad (t, t') \in T(PP \parallel QQ) \tag{4.19}$$

It has already been discussed that we only define the derived traces for the lifted forward behaviour. The lemma for lifted forward traces is defined as follows:

Lemma 4.11. $(PP \parallel QQ) \xrightarrow{t} R =$

$$\exists P, Q, p, q \cdot t \in (p \parallel q) \wedge PP \xrightarrow{p} P \wedge QQ \xrightarrow{q} Q \wedge R = (P \parallel Q)$$

The following equations are derived from the operational rules (3.20, 3.21) to support

the inductive proof of the above lemma.

$$
\begin{aligned}
(PP \parallel QQ) \xrightarrow{\omega} R \quad &= \quad PP \xrightarrow{\omega 1} P1 \;\wedge\; QQ \xrightarrow{\omega 2} Q \;\wedge\; \omega = \omega 1 \& \omega 2 \\
&\wedge \quad R = (P \parallel Q)
\end{aligned}
\tag{4.20}
$$

$$
\begin{aligned}
(PP \parallel QQ) \xrightarrow{a} RR \quad &= \quad PP \xrightarrow{a} PP' \;\wedge\; R = (PP' \parallel QQ) \\
&\vee \quad QQ \xrightarrow{a} QQ' \;\wedge\; R = (PP \parallel QQ')
\end{aligned}
\tag{4.21}
$$

The inductive proof of Lemma 4.11 can be found in the Appendix B. While the lemma is proved, it is now possible to prove the Equation 4.19. To prove the equation, we need the derived traces for both forward and compensation behaviour. Lemma 4.11 gives the lifted forward behaviour and we use Lemma 4.10 from standard processes to get the derived behaviour for the compensation.

Proof of Equation 4.19:

$$
\begin{aligned}
&\quad (t, t') \in DT(PP \parallel QQ) \\
&= (PP \parallel QQ) \xrightarrow{(t,t')} 0 \\
&= \exists R \cdot (PP \parallel QQ) \xrightarrow{t} R \;\wedge\; R \xrightarrow{t'} 0 \\
&= \text{``By Lemma 4.11''} \\
&\quad \exists P, Q, p, q \cdot t \in (p \parallel q) \;\wedge\; PP \xrightarrow{p} P \;\wedge\; QQ \xrightarrow{q} Q \;\wedge\; (P \parallel Q) \xrightarrow{t'} 0 \\
&= \text{``By Lemma 4.10''} \\
&\quad \exists P, Q, p, q \cdot (t \in (p \parallel q) \;\wedge\; PP \xrightarrow{p} P \;\wedge\; QQ \xrightarrow{q} Q \\
&\quad \wedge \exists p', q' \cdot t' \in (p' \parallel q') \;\wedge\; P \xrightarrow{p'} 0 \;\wedge\; Q \xrightarrow{q'} 0) \\
&= \text{``Remove } P, Q\text{''} \\
&\quad \exists p, p', q, q' \cdot t \in (p \parallel q) \;\wedge\; t' \in (p' \parallel q') \;\wedge\; PP \xrightarrow{p,p'} 0 \;\wedge\; QQ \xrightarrow{q,q'} 0 \\
&= \text{``Trace derivation rule''} \\
&\quad \exists p, p', q, q' \cdot t \in (p \parallel q) \;\wedge\; t' \in (p' \parallel q') \;\wedge\; (p, p') \in DT(PP) \;\wedge\; (q, q') \in DT(QQ) \\
&= \text{``Induction assumption''} \\
&\quad \exists p, p', q, q' \cdot t \in (p \parallel q) \;\wedge\; t' \in (p' \parallel q') \;\wedge\; (p, p') \in T(PP) \;\wedge\; (q, q') \in T(QQ) \\
&= \text{``By trace rules of parallel composition''} \\
&\quad (t, t') \in T(PP \parallel QQ) \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \square
\end{aligned}
$$

4.2.4 Compensation Pair

A compensation pair $(P \div Q)$ consists of two standard processes: a forward behaviour (P) and its compensation (Q). The semantics of compensation pair is defined in such a way that the behaviour of the compensation Q is augmented only with successfully completed forward behaviour of P, otherwise, the compensation is empty.

In order to show the correspondence between the two semantic models, in this section we show that,

$$(t, t') \in DT(P \div Q) \quad = \quad (t, t') \in T(P \div Q)$$

In the proof we assume that

$$DT(P) \quad = \quad T(P)$$
$$DT(Q) \quad = \quad T(Q)$$

The correspondence is proved by proving the following lemma, which is derived in the same way as the other operators from the definition of derived traces and the originally defined traces.

Lemma 4.12. $(P \div Q) \xrightarrow{(t,t')} 0 \;=\; \exists\, p, q \cdot (t, t') = (p \div q) \;\wedge\; P \xrightarrow{p} 0 \;\wedge\; Q \xrightarrow{q} 0$

We prove the lemma by using induction over traces. To support the inductive proof, we derived the following equations from the transition rules (3.23, 3.24, and 3.22) of operational semantics.

$$
\begin{aligned}
P \div Q \xrightarrow{\omega} R \;&=\; P \xrightarrow{\checkmark} 0 \;\wedge\; R = Q \\
&\quad\vee\; P \xrightarrow{\omega} 0 \;\wedge\; \omega \neq \checkmark \;\wedge\; R = \mathit{SKIP}
\end{aligned}
\tag{4.22}
$$

$$P \div Q \xrightarrow{a} RR \;=\; P \xrightarrow{a} P' \;\wedge\; RR = P' \div Q \tag{4.23}$$

Note that, unlike the other lemmas defined earlier for compensable processes, the lemma for the compensation pair is defined considering the traces of both forward and compensation behaviour. Consider the trace rules for the compensation pair, defined as follows:

$$
\begin{aligned}
\text{when } p = p'\langle\checkmark\rangle \text{ then} \quad &(t, t') \;=\; (p'\langle\checkmark\rangle \div q) \;=\; (p, q) \\
\text{when } p = p'\langle\omega\rangle \wedge \omega \neq \checkmark \text{ then} \quad &(t, t') \;=\; (p'\langle\omega\rangle \div q) \;=\; (p, \langle\checkmark\rangle)
\end{aligned}
$$

Supported by these trace rules and the above two equations, we prove the lemma for the compensation pair. The complete proof steps are placed in the Appendix B.

4.2.5   Transaction Block

A transaction block ($[\,PP\,]$) is a standard process where a compensable process (PP) is placed inside the block that converts it to a standard process. Hence, instead of having a pair of traces for the compensable process, we get a trace from a transaction block. Let t be the trace derived from a transaction block. Assuming $DT(PP) = T(PP)$, in

this section we show that,

$$t \in DT([PP]) \quad = \quad t \in T([PP])$$

From the definitions of derived and original traces, we state the following lemma to prove the semantic correspondence for the transaction block:

Lemma 4.13. $\quad [PP] \xrightarrow{t} 0 \; = \; \exists \, p, p' \cdot t = [p, p'] \; \wedge \; PP \xrightarrow{p,p'} 0$

In order to support the proof of Lemma 4.13 by using induction over traces, we derive the following equations from operational semantics. From rules 3.27, 3.28 (page 37) for terminal transition we drive the following equation:

$$
\begin{aligned}
[PP] \xrightarrow{\omega} 0 \; &= \; PP \xrightarrow{\omega} P \wedge \omega = \checkmark \\
&\vee \; PP \xrightarrow{!} P \wedge P \xrightarrow{\omega} 0
\end{aligned}
\tag{4.24}
$$

From transition rules 3.26 and 3.28 we derive the following equation:

$$
\begin{aligned}
[PP] \xrightarrow{a} R \; &= \; PP \xrightarrow{a} PP' \wedge R = [PP'] \\
&\vee \; PP \xrightarrow{!} P \wedge P \xrightarrow{a} R
\end{aligned}
\tag{4.25}
$$

We describe the inductive proof steps of the lemma as follows:

Proof:
Basic step: Case- $\langle \omega \rangle$

$\quad [PP] \xrightarrow{\omega} 0$

$=$ "From derived equation 4.24"

$\quad \exists \, P \cdot PP \xrightarrow{\omega} P \wedge \omega = \checkmark$ $\tag{4.26}$

$\vee \; \exists \, P \cdot PP \xrightarrow{!} P \wedge P \xrightarrow{\omega} 0$ $\tag{4.27}$

From equation (4.26)

$$\exists P \cdot PP \xrightarrow{\omega} P \wedge \omega = \checkmark$$

$=$ "By using Theorem 4.4"

$$\exists P \cdot PP \xrightarrow{\omega} P \wedge \omega = \checkmark \wedge \exists p' \cdot P \xrightarrow{p'} 0$$

$= \exists P, p, p' \cdot p = \langle \omega \rangle \wedge \omega = \checkmark \wedge PP \xrightarrow{p} P \wedge P \xrightarrow{p'} 0$

$=$ "By trace rule: $[\langle \omega \rangle, p'] = \langle \omega \rangle$ when $\omega = \checkmark$"

$$\exists P, p, p' \cdot p = \langle \omega \rangle \wedge \omega = \checkmark \wedge \langle \omega \rangle = [p, p'] \wedge PP \xrightarrow{p} P \wedge P \xrightarrow{p'} 0$$

$=$ "Removing P"

$$\exists p, p' \cdot \langle \omega \rangle = [p, p'] \wedge PP \xrightarrow{p,p'} 0 \wedge p = \langle \omega \rangle \wedge \omega = \checkmark$$

From equation (4.27)

$$\exists P \cdot PP \xrightarrow{!} P \wedge P \xrightarrow{\omega} 0$$

$= \exists P, p, p' \cdot p = \langle ! \rangle \wedge p' = \langle \omega \rangle \wedge PP \xrightarrow{p} P \wedge P \xrightarrow{p'} 0$

$=$ "By trace rule: $[\langle ! \rangle, \langle \omega \rangle] = \langle \omega \rangle$"

$$\exists P, p, p' \cdot \langle \omega \rangle = [\langle ! \rangle, \langle \omega \rangle] \wedge p = \langle ! \rangle \wedge p' = \langle \omega \rangle \wedge PP \xrightarrow{p} P \wedge P \xrightarrow{p'} 0$$

$=$ "Removing P "

$$\exists p, p' \cdot \langle \omega \rangle = [p, p'] \wedge p = \langle ! \rangle \wedge p' = \langle \omega \rangle \wedge PP \xrightarrow{p,p'} 0$$

Inductive step: Case - $\langle a \rangle t$

$$[PP] \xrightarrow{\langle a \rangle t} 0$$

$= \exists R \cdot [PP] \xrightarrow{a} R \wedge R \xrightarrow{t} 0$

$=$ "By using derived equation (4.25)"

$$\exists PP' \cdot PP \xrightarrow{a} PP' \wedge [PP'] \xrightarrow{t} 0 \tag{4.28}$$

$$\vee \exists P' \cdot PP \xrightarrow{!} P \wedge P \xrightarrow{\langle a \rangle t} 0 \tag{4.29}$$

From equation (4.28)

$$\exists\, PP' \cdot PP \xrightarrow{a} PP' \wedge [\,PP'\,] \xrightarrow{t} 0$$

= "Inductive hypothesis"

$$\exists\, PP' \cdot PP \xrightarrow{a} PP' \wedge \exists\, p', p'' \cdot t = [\,p'', p'\,] \wedge PP' \xrightarrow{p'', p'} 0$$

= "Removing PP'"

$$\exists\, p', p'' \cdot t = [\,p'', p'\,] \wedge PP \xrightarrow{\langle a\rangle p'', p'} 0$$

$$= \exists\, p, p', p'' \cdot p = \langle a\rangle p'' \wedge t = [\,p'', p'\,] \wedge PP \xrightarrow{p, p'} 0$$

= "By trace rule: $t = [\,p'', p'\,] \Rightarrow \langle a\rangle t = [\,\langle a\rangle p'', p'\,]$"

$$\exists\, p, p', p'' \cdot p = \langle a\rangle p'' \wedge \langle a\rangle t = [\,\langle a\rangle p'', p'\,] \wedge PP \xrightarrow{p, p'} 0$$

$$= \exists\, p, p', p'' \cdot p = \langle a\rangle p'' \wedge \langle a\rangle t = [\,p, p'\,] \wedge PP \xrightarrow{p, p'} 0$$

$$= \exists\, p, p' \cdot \langle a\rangle t = [\,p, p'\,] \wedge p \neq \langle\rangle \wedge PP \xrightarrow{p, p'} 0$$

From equation (4.29)

$$\exists\, P \cdot PP \xrightarrow{!} P \wedge P \xrightarrow{\langle a\rangle t} 0$$

$$= \exists\, P, p, p' \cdot p = \langle\rangle \wedge p' = \langle a\rangle t \wedge PP \xrightarrow{p\langle!\rangle} P \wedge P \xrightarrow{p'} 0$$

= "Removing P"

$$\exists\, p, p' \cdot p = \langle\rangle \wedge p' = \langle a\rangle t \wedge PP \xrightarrow{p\langle!\rangle, p'} 0$$

= "By trace rule: $[\,p\langle!\rangle, p'\,] = p, p'$"

$$\exists\, p, p' \cdot \langle a\rangle t = [\,p\langle!\rangle, p'\,] \wedge p = \langle\rangle \wedge p' = \langle a\rangle t \wedge PP \xrightarrow{p\langle!\rangle, p'} 0$$

$$= \exists\, p, p' \cdot \langle a\rangle t = [\,p\langle!\rangle, p'\,] \wedge p = \langle\rangle \wedge PP \xrightarrow{p\langle!\rangle, p'} 0 \qquad \square$$

4.2.6 Choice

Standard Processes

For standard processes P and Q, we show the correspondence between the semantic models by showing that,

$$t \in DT(P \,\square\, Q) \;=\; t \in T(P \,\square\, Q)$$

Following similar approach to other operators, we define the following lemma in order to prove the correspondence:

Lemma 4.14. $(P \,\square\, Q) \xrightarrow{t} 0 \;=\; \exists\, p, q \cdot t = (p \,\square\, q) \wedge P \xrightarrow{p} 0 \wedge Q \xrightarrow{q} 0$

Following equations are derived from operational semantics of the choice operator for standard processes to support the inductive proof of the above lemma:

$$
\begin{aligned}
(P \,\Box\, Q) \xrightarrow{\omega} 0 \;\;&=\;\; P \xrightarrow{\omega} 0 \\
&\;\;\vee\;\; Q \xrightarrow{\omega} 0
\end{aligned}
\tag{4.30}
$$

$$
\begin{aligned}
(P \,\Box\, Q) \xrightarrow{a} R \;\;&=\;\; P \xrightarrow{a} P' \;\wedge\; R = P' \\
&\;\;\vee\;\; Q \xrightarrow{a} Q' \;\wedge\; R = Q'
\end{aligned}
\tag{4.31}
$$

The inductive proof of the choice operator is very trivial. By using the above two derived equations and Theorem 4.4, we can prove both basic and inductive step of the lemma. The proofs are omitted from the description here.

Compensable Processes

For compensable processes PP and QQ, we prove the following equation:

$$
(t, t') \in DT(PP \,\Box\, QQ) \;\;=\;\; (t, t') \in T(PP \,\Box\, QQ)
$$

We only derive traces for the lifted forward behaviour and define the following lemma for the lifted forward traces:

Lemma 4.15. $(PP \,\Box\, QQ) \xrightarrow{t} R \;=$
$$\exists\, p, q \cdot t = (p \,\Box\, q) \;\wedge\; PP \xrightarrow{p} P \;\wedge\; QQ \xrightarrow{q} Q \;\wedge\; R = (P \,\Box\, Q)$$

To support the inductive proof of the lemma we derive the following equations from the operational semantics:

$$
\begin{aligned}
(PP \,\Box\, QQ) \xrightarrow{\omega} R \;\;&=\;\; PP \xrightarrow{\omega} P \;\wedge\; R = P \\
&\;\;\vee\;\; QQ \xrightarrow{\omega} Q \;\wedge\; R = Q
\end{aligned}
\tag{4.32}
$$

$$
\begin{aligned}
(PP \,\Box\, QQ) \xrightarrow{a} RR \;\;&=\;\; PP \xrightarrow{a} PP' \;\wedge\; RR = PP' \\
&\;\;\vee\;\; QQ \xrightarrow{a} QQ' \;\wedge\; RR = QQ'
\end{aligned}
\tag{4.33}
$$

As the inductive proof is trivial, it is omitted from the presentation here.

SKIP, *THROW* and *YIELD* are the primitive processes of cCSP. The compensable counterparts of the primitive processes are *SKIPP*, *THROWW* and *YIELDD* respectively and they are defined by using the pairing operator. The correspondence proofs of these primitive processes are also omitted from here.

4.2.7   Role of Trace operator

The trace operators play a significant role in defining the lemmas as well as in the correspondence proofs. The operators are used both at the trace levels and the process levels. All the lemmas defined in this chapter have a common pattern applicable to both standard and compensable processes. For example, for standard processes P and Q, and their traces p and q, the lemmas for all the operators are defined as follows:

$$(P \otimes Q) \xrightarrow{t} 0 \;=\; \exists\, p, q \cdot t = (p \otimes q) \;\wedge\; P \xrightarrow{p} 0 \;\wedge\; Q \xrightarrow{q} 0$$

$$\text{(for parallel operator use } t \in (p \otimes q) \text{ instead of } t = (p \otimes q))$$

Similar definitions are also given for the forward behaviour of compensable processes. The use of operators at both trace and process levels allow us to apply appropriate rules for the operators (rules for terminal and observable events from operational and trace semantics).

Such application of trace operators simplifies the proof steps considerably in our inductive proofs. For example, at the early stage of our proofs, by following the approach given in [59] for sequential composition, we defined the Lemma 4.6 as follows:

$$(P \; ; \; Q) \xrightarrow{t\langle\omega\rangle} 0 \;\;=\;\; \exists\, p, q \cdot P \xrightarrow{p\langle\checkmark\rangle} 0 \;\wedge\; Q \xrightarrow{q} 0 \;\wedge\; t = p.q$$
$$\vee \quad P \xrightarrow{t\langle\omega\rangle} 0 \;\wedge\; \omega \neq \checkmark$$

While proving by applying induction over traces, the above definition led us to an explosion of several sub-cases and deriving the proofs became complicated. Similar conditions also apply to other operators. The present definition not only simplifies the definition of lemmas, but also minimises the explosion of sub-cases in the inductive proofs.

4.3   Related Work

The semantic correspondence presented here is based on the technique of applying structural induction. A similar approach is also applied by S. Schneider [105], where an equivalence relation was established between the operational and denotational semantics of timed CSP [101, 106]. Operational rules are defined for timed CSP and then timed traces and refusals are extracted from the transition rules of a program, and it is shown that the pertinent information corresponds to the semantics obtained from the denotational semantic function. By applying structural induction over the terms of timed CSP, it was proved that the behaviour of the transition system is identical to those provided by the denotational semantics.

A similar problem was also investigated in [118], where a metric structure was employed to relate the operational and denotational models of a given language. In order to relate the semantic models it was proved that the two models coincide. The denotational models were extended and structural induction was applied over the terms of the language to relate the semantic models.

Other than using induction, Hoare and He [60] presented the idea of unifying different programming paradigms and showed how to derive operational semantics from its denotational presentation of a sequential language. They derive algebraic laws from the denotational definition and then derive the operational semantics from the algebraic laws. Similar to our work, Huibiao *et al.* [125] derived denotational semantics from operational semantics for a subset of Verilog [51]. However the derivation was done in a different way than our method where the authors defined transitional condition and phase semantics from the operational semantics. The denotational semantics are derived from the sequential composition of the phase semantics. The authors also derived operational semantics from denotational semantics [124].

Unlike our approach, the unification between the two semantics was shown in [111] by extending the operational semantics to incorporate the denotational properties. The equivalence was shown for a language having simple models without any support for concurrency. Similar problem was also investigated in [74] for a simple sequential language, which support recursion and synchronisation in the form of interleaving. The relation between operational and denotational semantics is obtained via an intermediate semantics.

4.4   Discussion

Two key factors play a vital role in deriving the correspondence proofs. One is the use of operators at the trace level (discussed in Section 4.2.7) and another is the derivation of rules from the transition rules of the operators. For each operator, two equations are derived from the operational rules: one is used in the base case and the other is used in the inductive case of the proofs.

We have adopted a systematic approach to show the correspondence between the two semantic models. We defined how to derive traces from the transition rules in the operational semantics. For each term of the language, separate lemmas are defined to show the correspondence between the derived traces with the originally defined traces. The lemmas are proved by applying induction over individual traces. The lemmas form a common pattern for all the terms of the language. The semantic correspondence is established by applying structural induction over the process terms.

The traces of a process are represented by a sequence of observable events followed by

a terminal event. The nature of the traces is determined by the final terminal events ($\checkmark, !, ?$). There are also two types of transition rules defined in operational semantics labelled by normal and terminal events. Having separate symbols as labels allow us to extract traces for both normal and terminal events that helps to carry out the correspondence proofs shown in this section.

Two levels of induction have been applied in the correspondence proofs. At one level induction is applied over traces and at another level induction is applied on process terms. For example, Lemma 4.6 was for sequential operator for individual traces. It was then lifted to the set of traces to prove the correspondence.

The correspondence proofs presented in this chapter are completely performed by hand which has a strong possibility to introduce several errors in the proofs as well as to overlook some important features. Chapter 6 is devoted to showing the mechanisation of the semantic models as well as their relationship by using the theorem prover PVS.

In the semantic description of cCSP, presented so far, we have avoided the synchronous communication of processes on observable events. Synchronised communication is an important and well understood feature of process algebras for distributed and concurrent systems. The semantics of the synchronised parallel processes are presented in Chapter 5 along with the correspondence proofs.

Chapter 5

Extending the Semantics to Synchronisation

Synchronisation is an important feature for concurrent and distributed processes. In the semantic definition given in earlier chapters, processes in parallel composition do not synchronise over observable events. In this chapter, we extend the semantic models to support synchronisation between processes over observable events.

5.1   Introduction

We extend the operational semantics and define the semantics for synchronisation. While defining synchronisation we introduce a new terminal event that supports the semantic definition of synchronisation. In this chapter we define an operational semantics for the synchronised parallel operator.

The synchronisation between processes can lead to a deadlock situation, where no further execution of events is possible. From a synchronous composition that leads to deadlock, it is not possible to get a complete behaviour, instead we get partial traces of behaviour. Before defining the semantics of synchronous composition, we define the notion of *partial behaviour*. We show how the notion of partial behaviour is added to the existing semantic definitions and how it allows us to define the correspondence for the synchronous semantic models.

In the rest of the chapter, first we define the notion of partial behaviour in Section 5.2. In Section 5.3 we briefly outline the trace semantics for the synchronised parallel operator (see details in [26]). We define the transition rules for the synchronous composition of both standard and compensable processes in Section 5.4. The correspondence proofs are then outlined in Section 5.5. Finally, Section 5.6 summarises the results of the chapter.

5.2   Partial behaviour

We assume a special terminal symbol $\perp \in \Omega$ which indicates a partial trace. Partial traces are analogous to prefixes in standard CSP. Partial traces with respect to a set of traces of a process have the following property:

$$p\langle x \rangle q \in T(P) \;\; \Rightarrow \;\; p\langle \perp \rangle \in T(P)$$

We assume that $\perp$ acts as a cut for trace concatenation:

$$p\langle \perp \rangle q \;=\; p\langle \perp \rangle$$

Processes now have at least a trace of the form:

$$\langle \perp \rangle \in T(P)$$

In parallel composition of processes, some events will synchronise and others will be interleaved. We assume a set of events X over which processes will synchronise. To model synchronisation between processes we define a synchronisation operator on events writing $A \,\&\, A'$ for the synchronisation of events A and A'. Consider two processes synchronise over events a and a', the synchronisation is defined as follows:

$$
\begin{aligned}
a \,\&\, a &= a \\
a \,\&\, a' &= \perp \quad (a \neq a')
\end{aligned}
$$

Here we are assuming that although both a and a' are synchronising events, they do not synchronise with each other.

In cCSP, the process $P \parallel_X Q$ represents the parallel composition of two processes P and Q synchronising over the set of events X. Operationally, P and Q interact by synchronising over the events from X, while events not in X can occur independently. An event where both processes synchronise becomes a single event in $P \parallel_X Q$, which is defined above by the synchronising operator. We show the effect of $\perp$ on the behaviour of synchronising processes in the following sections.

The synchronisation between a $\perp$ and a terminal event ω results in a $\perp$, defined as follows:

$$\perp \,\&\, \omega \;=\; \perp$$

Based on the definition given above for partial behaviour and the synchronising operator, in the rest of the chapter, we first give the definition of trace semantics for concurrency operator of synchronising processes (see details in [26]). The main contribution of this

chapter starts by defining the transition relation to define the operational semantics for the concurrency operator. Following the definition of transition relations for both standard and compensable processes, we first define the correspondence relationship between the semantic models and then outline the way to prove the correspondence between the two semantic models for synchronising concurrency operator. The detailed steps of the proofs are not shown here. Later in Chapter 6, we will show the mechanisation of the correspondence proof.

5.3 Trace Semantics

With the introduction of $\perp$, processes can have two types of traces: completed trace and partial trace. A completed trace ends with a terminal event other than a $\perp$ ($\checkmark, !, ?$) and a partial trace ends with a $\perp$.

5.3.1 Standard Processes

After the introduction of partial behaviour, the traces of a standard process P have one of the following forms:

$$
\text{Completed trace: } p\langle\omega\rangle \quad = \quad
\begin{cases}
p\langle\checkmark\rangle & \text{successful termination} \\
p\langle!\rangle & \text{termination with interrupt throw} \\
p\langle?\rangle & \text{termination with interrupt yield}
\end{cases}
$$

$$
\text{Partial trace: } p\langle\perp\rangle \quad = \quad \text{Partial behaviour}
$$

Standard processes are defined as a non-empty set of such traces. Standard processes now have at least a trace of the form: $\langle\perp\rangle \in T(P)$.

We define the trace semantics by defining the synchronising operator on traces and then lift it to processes.

Definition 5.1 (Parallel composition of traces). Parallel composition of two traces with respect to the synchronisation set X is a set of traces. Assume $a, a' \in X$, i.e., a and a' are in the synchronisation set and $b, b' \notin X$. The parallel composition of traces

is defined as follows:

$$\langle\omega\rangle \quad \|_X \quad \langle\omega'\rangle \quad = \quad \{\langle\omega \,\&\, \omega'\rangle\} \ \textbf{where} \quad \begin{array}{c|cccccc} \omega & ! & ! & ! & ? & ? & \checkmark & \perp \\ \omega' & ! & ? & \checkmark & ? & \checkmark & \checkmark & \omega \\ \hline \omega \,\&\, \omega' & ! & ! & ! & ? & ? & \checkmark & \perp \end{array}$$

$$\langle a\rangle p \quad \|_X \quad \langle\omega\rangle \quad = \quad \{\langle\perp\rangle\}$$

$$\langle\omega\rangle \quad \|_X \quad \langle a\rangle q \quad = \quad \{\langle\perp\rangle\}$$

$$\langle a\rangle p \quad \|_X \quad \langle a'\rangle q \quad = \quad \{\langle a''\rangle r \mid a'' \in (a \,\&\, a') \,\wedge\, r \in (p \|_X q)\}$$

$$\langle b\rangle p \quad \|_X \quad \langle\omega\rangle \quad = \quad \{\langle b\rangle r \mid r \in (p \|_X \langle\omega\rangle)\}$$

$$\langle\omega\rangle \quad \|_X \quad \langle b\rangle q \quad = \quad \{\langle b\rangle r \mid r \in (\langle\omega\rangle \|_X q)\}$$

$$\langle b\rangle p \quad \|_X \quad \langle a\rangle q \quad = \quad \{\langle b\rangle r \mid r \in (p \|_X \langle a\rangle q)\}$$

$$\langle a\rangle p \quad \|_X \quad \langle b\rangle q \quad = \quad \{\langle b\rangle r \mid r \in (\langle a\rangle p \|_X q)\}$$

$$\langle b\rangle p \quad \|_X \quad \langle b'\rangle q \quad = \quad \{\langle b\rangle r \mid r \in (p \|_X \langle b'\rangle q)\} \ \cup \ \{\langle b'\rangle r \mid r \in (\langle b\rangle p \|_X q)\}$$

Traces of a parallel composition of processes are defined in terms of this trace definition.

Definition 5.2 (**Parallel composition of processes**)**.** The trace semantics of a synchronous parallel composition is defined as a set of traces of the form,

$$T(P \|_X Q) \quad = \quad \{r \mid r \in (p \|_X q) \,\wedge\, p \in T(P) \,\wedge\, q \in T(Q)\}$$

5.3.2 Compensable Processes

Traces of a compensable processes consist of a set of pairs of traces of the form $(p\langle\omega\rangle, p'\langle\omega'\rangle)$, where $p\langle\omega\rangle$ represents the forward behaviour and $p'\langle\omega'\rangle$ represents the compensation. With the definition of partial behaviour ($\perp$), the pair of traces follow the following rules:

- $\omega = \perp \ \Rightarrow\ p'\langle\omega'\rangle = \langle\perp\rangle$

- $(p\langle x\rangle q, p') \in T(PP) \ \Rightarrow\ (p\langle\perp\rangle, \langle\perp\rangle) \in T(PP)$

Taking into account the above rules for pairs of traces, we define the parallel composition of pairs of traces as follows:

Definition 5.3 (**Compensable parallel composition of traces**)**.**

$$\begin{aligned}(p, p') \|_X (q, q') \quad = \quad & \{(r, r') \mid r \in (p \|_X q) \,\wedge\, r' \in (p' \|_X q') \wedge last(r) \neq \perp\} \\ \cup \quad & \{(r, \langle\perp\rangle) \mid r \in (p \|_X q) \,\wedge\, last(r) = \perp\}\end{aligned}$$

We use an operator over traces: $last(t)$, which returns the terminal symbol from a trace t. The above definition consists of two parts; the first part does not contain any

partial behaviour in the forward traces ($last(r) \neq \perp$), and the second part is considering partial behaviour in the forward traces ($last(r) = \perp$).

Traces of compensable processes are defined as a set of such pairs of traces, defined as follows:

Definition 5.4 (Compensable parallel composition).

$$T(PP \parallel_X QQ) \quad = \quad \{\, rr \mid rr \in (pp \parallel_X qq) \wedge pp \in T(PP) \wedge qq \in T(QQ) \,\}$$

In the next section, we define the operational semantics of the synchronised parallel operator for both standard and compensable processes.

5.4   Operational semantics

The transition relations defined in Chapter 3 are extended by adding the transition of process terms with the newly defined $\perp$ event. When the synchronising events from separate parallel processes do not synchronise with each other, we represent it by the bottom event, $\perp$. In this case, we define the transitions for both standard and compensable processes, where both processes terminate to a null process. For example, consider two events A and A', where A synchronises over a, and A' synchronises on a', and a and a' do not synchronise with each other. Then the parallel composition of A and A' is defined as,

$$(A \parallel_{\{a,a'\}} A') \xrightarrow{\perp} 0$$

In the following sections, we present the operational semantics of synchronised parallel composition for both standard and compensable processes. For both types of processes, we will describe the effect of the $\perp$ event in transition rules.

5.4.1   Standard Processes

The transition relation for standard processes is extended by adding the transition of a process term by a $\perp$, as follows:

$$P \xrightarrow{\perp} 0 \tag{5.1}$$

We define the transitions of a standard process term by synchronising observable events. We also define the transitions where events from separate concurrent processes introduce a $\perp$ and this results in a transition by a $\perp$. Although $\perp$ is not operationally meaningful, it is a useful semantic device that helps us in solving the semantic correspondence.

Hence, we define the transition rules that introduce a $\bot$ in their transitions. We define the transition rules for both synchronous and asynchronous events.

– Concurrently executing processes synchronises over terminal events. The synchronisation of two terminal events ($\omega \,\&\, \omega'$) is defined in the previous section and any of the terminal events can be the $\bot$ event.

$$\frac{P \xrightarrow{\;\omega\;} 0 \;\wedge\; Q \xrightarrow{\;\omega'\;} 0}{P \,\|_X\, Q \xrightarrow{\;\omega\&\omega'\;} 0} \quad (\omega, \omega' \in \Omega) \tag{5.2}$$

– The synchronisation between a terminal event and a synchronising event results in a $\bot$ event and the composition terminates with a null process. The bottom event here indicates the partial behaviour of the composition.

$$\frac{P \xrightarrow{\;a\;} P' \;\wedge\; Q \xrightarrow{\;\omega\;} 0}{P \,\|_X\, Q \xrightarrow{\;\bot\;} 0} \qquad \frac{P \xrightarrow{\;\omega\;} 0 \;\wedge\; Q \xrightarrow{\;a\;} Q'}{P \,\|_X\, Q \xrightarrow{\;\bot\;} 0} \quad (a \in X,\; \omega \in \Omega) \tag{5.3}$$

– Parallel processes synchronise over events from the set of synchronising events X and the transition is labelled by the synchronisation of the events. When the processes fail to synchronise then it results the transition to a null process.

$$\frac{p \xrightarrow{\;a\;} P' \;\wedge\; Q \xrightarrow{\;a'\;} Q'}{P \,\|_X\, Q \xrightarrow{\;a\&a'\;} P' \,\|_X\, Q'} \quad (a, a' \in X) \tag{5.4}$$

– For non synchronising events, processes interleave with each other.

$$\frac{P \xrightarrow{\;b\;} P'}{P \,\|_X\, Q \xrightarrow{\;b\;} P' \,\|_X\, Q} \qquad \frac{Q \xrightarrow{\;b\;} Q'}{P \,\|_X\, Q \xrightarrow{\;b\;} P \,\|_X\, Q'} \quad (b \notin X) \tag{5.5}$$

5.4.2 Compensable Processes

We show the effect of the $\bot$ (bottom) event on compensable processes and show how it affects the compensations. We extend the transition rules of compensable processes in order to define the transitions by the $\bot$ event. Unlike the usual terminal transitions (by $\checkmark, !, ?$), where a compensable process terminates and the attached compensations are accumulated, no compensation is accumulated from the transition by a $\bot$. The transition by a $\bot$ for a compensable process PP is defined as follows:

$$PP \xrightarrow{\;\bot\;} 0$$

Like for standard processes, we also define the transition rules for those cases where events from separate concurrent processes introduce the $\bot$ and then transitions of process terms are defined according to the transition rule defined above.

– In a synchronous composition, processes synchronise on terminal events but as the processes are compensable, instead of terminating with a null process, corresponding compensations from the processes will be accumulated in parallel. Note that $\perp$ is excluded in this rule.

$$\frac{PP \xrightarrow{\omega} P \ \wedge \ QQ \xrightarrow{\omega'} Q}{PP \parallel_X QQ \xrightarrow{\omega \,\&\, \omega'} P \parallel_X Q} \quad (\omega \,\&\, \omega' \neq \perp) \tag{5.6}$$

– If the synchronisation of the terminal events results in a $\perp$, then unlike with the usual terminal events where the forward behaviour of compensable processes terminate leaving the compensation to be stored for future reference, the concurrently executing processes terminate to a null process. The $\perp$ event here indicates the partial behaviour of the composition.

$$\frac{PP \xrightarrow{\omega} P \ \wedge \ QQ \xrightarrow{\omega'} Q}{PP \parallel_X QQ \xrightarrow{\omega \,\&\, \omega'} 0} \quad (\omega \,\&\, \omega' = \perp) \tag{5.7}$$

– The synchronisation between a terminal event and events from the synchronisation set X also introduce the $\perp$ event and the compensable processes terminate to a null process representing the partial behaviour of the composition.

$$\frac{PP \xrightarrow{a} PP' \ \wedge \ QQ \xrightarrow{\omega} Q}{PP \parallel_X QQ \xrightarrow{\perp} 0} \quad \frac{PP \xrightarrow{\omega} P \ \wedge \ QQ \xrightarrow{a} QQ'}{PP \parallel_X QQ \xrightarrow{\perp} 0} \ (a \in X, \, \omega \in \Omega) \tag{5.8}$$

– Processes synchronise on events in set X and the synchronised events cause the transition of both processes from one state to another and the transition is labelled by the synchronisation of the events. Failure to synchronise over the events from the set X introduce the $\perp$ and it results the transition to a null process.

$$\frac{PP \xrightarrow{a} PP' \ \wedge \ QQ \xrightarrow{a'} QQ'}{PP \parallel_X QQ \xrightarrow{a \,\&\, a'} PP' \parallel_X QQ'} \quad (a, a' \in X) \tag{5.9}$$

– For non-synchronising events, processes interleave with each other.

$$\frac{PP \xrightarrow{b} PP'}{PP \parallel_X QQ \xrightarrow{b} PP' \parallel_X QQ} \quad \frac{QQ \xrightarrow{b} QQ'}{PP \parallel_X QQ \xrightarrow{b} PP \parallel_X QQ'} \quad (b \notin X) \tag{5.10}$$

After defining the trace and operational semantics of synchronising parallel processes for cCSP, in the next section, we define the relationship between the semantic models and then outline the correspondence proofs. The outline for the proofs is presented for both standard and compensable processes. For both processes, theorems and supporting lemmas are drawn. We have already mentioned that we do not show the proof steps as we prove the correspondence by using the theorem prover PVS instead in Section 6.

5.5   Semantic Correspondence

In this section, we show the correspondence between the operational and trace semantics of synchronous parallel composition. The approach taken here to prove the correspondence is the same as that in the previous chapter with the newly defined bottom ($\perp$) event added.

In Chapter 4, we showed the definition of derived traces for both standard and compensable processes. With the introduction of partial behaviour, the definition of derived traces remains the same except for the compensable processes.

Compensable processes consist of forward and compensation behaviour and the behaviour of compensation is similar to standard processes. Consider a compensable process PP, and t and t' are the forward and compensation traces, then

$$PP \xrightarrow{(t,t')} 0 \;=\; \begin{cases} \exists\, R \cdot PP \xrightarrow{\;t\;} R \;\wedge\; R \xrightarrow{\;t'\;} 0 & last(t) \neq \perp \\ PP \xrightarrow{\;t\;} 0 \;\wedge\; t' = \langle \perp \rangle & last(t) = \perp \end{cases}$$

The operator $last(t)$ returns the terminal symbol from the trace t.

By using this definition we get the trace derivation rule of a compensable process PP, defined as follows:

Definition 5.5 (**Derived trace: Compensable process**). For traces t and t',

$$(t, t') \in DT(PP) \quad=\quad PP \xrightarrow{(t,t')} 0$$

Considering the theorem (Theorem 4.3) defined in Chapter 4, in page 47, in this section we prove the theorem only for the synchronising operator, defined as follows:

Theorem 5.6. *For standard non-null process terms P and Q, assuming $DT(P) = T(P)$ and $DT(Q) = T(Q)$,*

$$DT(P \parallel_X Q) \quad=\quad T(P \parallel_X Q)$$

For non-null compensable process terms PP and QQ, assuming $DT(PP) = T(PP)$ and $DT(QQ) = T(QQ)$,

$$DT(PP \parallel_X QQ) \quad=\quad T(PP \parallel_X QQ)$$

The theorems 4.4 and 4.5 defined in Chapter 4 are also applicable to the synchronised parallel operator.

5.5.1 Standard Processes

Following the same approach as for asynchronous parallel composition, we derive the following lemma in order to establish the required correspondence:

Lemma 5.7. $(P \parallel_X Q) \xrightarrow{t} 0 = \exists p, q \cdot t \in (p \parallel_X q) \wedge P \xrightarrow{p} 0 \wedge Q \xrightarrow{q} 0$

The lemma is proved by using induction over traces. To support the induction scheme, the following equations are derived from the operational rules, categorised according to the events by which the transition rules are defined.

i) $(P \parallel_X Q) \xrightarrow{a} R$ (from rule 5.4)

   $= \exists a1, a2, P', Q' \cdot a = a1 \& a2 \wedge P \xrightarrow{a1} P' \wedge Q \xrightarrow{a2} Q' \wedge R = (P' \parallel_X Q')$

ii) $(P \parallel_X Q) \xrightarrow{b} R$ (from rule 5.3)

   $= \exists b, P' \cdot P \xrightarrow{b} P' \wedge R = (P' \parallel_X Q)$

   $\vee \exists b, Q' \cdot Q \xrightarrow{b} Q' \wedge R = (P \parallel_X Q')$

iii) $(P \parallel_X Q) \xrightarrow{\omega} 0$ $(\omega \neq \perp)$ (from rule 5.2)

   $= \exists \omega1, \omega2 \cdot \omega = \omega1 \& \omega2 \wedge P \xrightarrow{\omega1} 0 \wedge Q \xrightarrow{\omega2} 0$

iv) $(P \parallel_X Q) \xrightarrow{\perp} 0$ (from ules 5.3, 5.4)

   $= \exists a, P', \omega' \cdot P \xrightarrow{a} P' \wedge Q \xrightarrow{\omega'} 0$

   $\vee \exists a, Q', \omega' \cdot Q \xrightarrow{a} Q' \wedge P \xrightarrow{\omega'} 0$

   $\vee \exists a1, a2, P', Q' \cdot P \xrightarrow{a1} P' \wedge Q \xrightarrow{a2} Q' \wedge a1 \& a2 = \perp$

Proof: A sketch of the proof is given here

Basic Step:

In the basic step, we prove the lemma for the case:

$$(P \parallel_X Q) \xrightarrow{\langle \omega \rangle} 0$$

In order to prove the basic case, the derived equations (iii) and (iv) are to be considered.

Inductive Step:

In the inductive step, two cases are to be considered:

- *synchronous*$(\langle a \rangle t)$: $(P \parallel_X Q) \xrightarrow{\langle a \rangle t} 0$

- *asynchronous* $(\langle b \rangle t)$: $(P \parallel_X Q) \xrightarrow{\langle b \rangle t} 0$

In order to prove the inductive case, the derived equation

(i) is considered for the synchronous case and

(ii) is considered for the asynchronous cases.

5.5.2 Compensable Processes

The compensation behaviour of a compensable process is similar to a standard process and we can reuse the above proof for the compensation behaviour. When the concurrent compensable processes do not synchronise on some synchronising events, we get a partial behaviour from the composition and no attached compensation is returned from the composition. Considering this scenario, we state two separate lemmas.

First, we state a lemma assuming that there is no failure during the synchronisation of processes:

Lemma 5.8. $(PP \parallel_X QQ) \xrightarrow{t} R \quad = \quad \exists\, p, q, P, Q \cdot t \in (p \parallel_X q) \wedge last(t) \neq \perp$

$$\wedge \quad PP \xrightarrow{p} P \wedge QQ \xrightarrow{q} Q \wedge R = (P \parallel_X Q)$$

The following lemma is defined for the cases when the synchronising processes fail to synchronise during their composition:

Lemma 5.9. $(PP \parallel_X QQ) \xrightarrow{t} 0 \quad = \quad \exists\, p, q \cdot t \in (p \parallel_X q) \wedge last(t) = \perp$

$$\wedge \quad p \in T(PP) \wedge q \in T(QQ)$$

Both lemmas are proved by using induction. In order to support the proofs of the lemmas, we derive equations from the operational semantics as for the previous lemmas. The derived equations are defined as follows:

i) $(PP \parallel_X QQ) \xrightarrow{a} RR \qquad$ (from rule 5.9)
$= \exists\, PP', QQ', a1, a2 \cdot a = a1\,\&\,a2 \wedge PP \xrightarrow{a1} PP' \wedge QQ \xrightarrow{a2} QQ'$
$\wedge \; RR = (PP' \parallel_X QQ')$

ii) $(PP \parallel_X QQ) \xrightarrow{b} RR \qquad$ (from rule 5.10)
$= \exists\, PP' \cdot PP \xrightarrow{b} PP' \wedge RR = (PP' \parallel_X QQ)$
$\vee \, \exists\, QQ' \cdot QQ \xrightarrow{b} QQ' \wedge RR = (PP \parallel_X QQ')$

iii) $(PP \parallel_X QQ) \xrightarrow{\omega} R \;\; (\omega \neq \perp) \qquad$ (from rule 5.6)
$= \exists\, P, Q, \omega 1, \omega 2 \cdot \omega = \omega 1\,\&\,\omega 2 \wedge \omega 1\,\&\,\omega 2 \neq \perp$
$\wedge \; PP \xrightarrow{\omega 1} P \wedge QQ \xrightarrow{\omega 2} Q \wedge R = (P \parallel_X Q)$

iv) $(PP \parallel_X QQ) \xrightarrow{\perp} 0 \qquad$ (from rules 5.7, 5.8, 5.9)
$= \exists\, PP', Q, \omega \cdot PP \xrightarrow{a} PP' \wedge QQ \xrightarrow{\omega} Q$
$\vee \, \exists\, QQ', P, \omega \cdot QQ \xrightarrow{a} QQ' \wedge PP \xrightarrow{\omega} P$
$\vee \, \exists\, PP', QQ', a1, a2 \cdot PP \xrightarrow{a1} PP' \wedge QQ \xrightarrow{a2} QQ' \wedge a1\&a2 = \perp$
$\vee \, \exists\, P, Q, \omega 1, \omega 2 \cdot PP \xrightarrow{\omega 1} P \wedge QQ \xrightarrow{\omega 2} Q \wedge \omega 1\,\&\,\omega 2 = \perp$

Proof: A sketch of the proof is given here

Basic Step:

– For Lemma 5.8, we consider the following case,

$$(PP \parallel_X QQ) \xrightarrow{\omega} R \quad (\omega \neq \bot)$$

Derived equation (iii) is to be considered for this case.

– For Lemma 5.9, we consider

$$(PP \parallel_X QQ) \xrightarrow{\bot} 0$$

We use the derived equation (iv) for this case.

Inductive Step: In the inductive steps of the proofs, we need to consider two separate cases for each lemma:

– *Asynchronous:* In the asynchronous case, the derived equation (ii) is to be used for both of the above lemmas.

– *Synchronous:* In the case where processes synchronise with each other and the derived equation (i) is taken into account in the inductive proofs of the above two lemmas.

5.6   Summary

In this chapter, we have extended the semantic models of cCSP by defining the operational semantics for synchronising processes. During synchronisation some synchronising events might fail to synchronise with each other and its semantics is defined by using the notion of partial behaviour ($\bot$) which is analogous to prefixes in standard CSP. The $\bot$ acts as a final terminal symbol of the traces of a deadlocked process. It allows partial behaviour to be extracted from a synchronous composition that lead to deadlock.

We have also outlined how to derive the correspondence between the trace and operational semantics. In contrast to our earlier approach (Chapter 4), we have avoided describing the proof steps in detail. Next chapter shows the mechanisation of the semantic models and their relationship by using the theorem prover PVS.

Chapter 6

Mechanising Semantic Models and their Relationships

6.1   Introduction

A way to combine the strength of general purpose theorem provers with formal notations is the semantic embedding of the formal notations within the logic of a verification system. PVS [89] is an automated framework for specification and verification. In this chapter, we investigate how the semantic models of the cCSP process algebra can be incorporated in the framework of the PVS tool. The intention is to use PVS to prove semantic properties, especially the relationship between the semantic models of cCSP, with similar level of granularity of or even more precisely than, the hand proofs.

In Chapter 4, we have shown how to derive the correspondence between the two semantic models of cCSP. The proofs establishing the relationship have been carried out completely by hand. However, while doing the proofs by hand it is a difficult and tedious task to manage a large number steps and subtle mistakes or omissions can easily occur in any stage of the proof. A tool allowing one to mechanise the semantic models and to mechanically prove the correspondence can overcome the problem. Such a proof assistant gives confidence in the formulation of the semantics, and alleviates the burden of spelling out each of the intermediate steps, for human beings a tedious and extremely error-prone task. For this reason, we aim at the mechanisation of the semantic relationships of cCSP. An interactive/automatic theorem prover which can successfully mechanise our proofs, will demonstrate the correctness of the manual proofs and consequently, feasible mechanisation allows us to follow the same mechanical proof technique to apply to larger proofs.

The capabilities of general theorem provers to support formal reasoning have been increased in recent years. To address our problem, presently there are several systems such

as PVS [89], HOL [52], Isabelle [94] and Coq [11], having a rich specification language and automated support for decision procedures and proof strategies to their logic. One of our goals is to follow the style of proofs performed by hand. Some of the benefits of human-style proof are reported in [7]. PVS provides both a highly expressive language and automation of proof steps. PVS supports high-order logic, allows abstract datatypes to model process terms and has strong inductive support as well, which are the building blocks of our definitions. Existing research works such as [12, 38, 40] have been carried out to mechanise process algebra. The trace semantics of CSP have been mechanised in [38, 40]. The cCSP semantic models are closely related to that of CSP and given the positive experience described in [38, 40] we decided to use PVS to address our problems.

Properties of a process algebra can be proved in PVS by means of an interactive proof checker. The user applies proof commands to simplify the goals that must be proven, until it can be proved automatically by the powerful decision procedures of the tool. We define process terms by means of the abstract datatype mechanism of PVS, which generates a useful induction scheme allowing induction on the structure of the terms.

6.1.1 Chapter Contribution

The contribution of this chapter is mechanising the correspondence relationship between the operational and trace semantics of cCSP process algebra by using the theorem prover PVS. We are not aware of any similar mechanisation of the relationship between a denotational and operational semantics. Both [38] and [40] are only concerned with properties of the traces models. We mechanise the trace semantics and recursively define the transition rules of the operational semantics in PVS. The theorems and lemmas to derive the correspondence are mechanised and proved in PVS by following similar proof steps as the proof by hand where induction is applied not only over traces but also over process terms. We also show how the mechanisation of the relationship helps us to identify some of the important issues that are ignored in the hand proofs. Future extensions of the language could also benefit from such mechanisation where semantic relationships can be verified in a similar fashion without any hand proof.

6.1.2 Chapter Structure

The rest of the chapter is structured as follows. We give a brief overview of PVS datatypes that are being used in our specification in Section 6.2. Section 6.3 describes the embedding of syntax and process terms of cCSP. The mechanisation of the trace semantics is described in Section 6.4 and the mechanisation of the operational semantics is described in the following section. Section 6.6 shows how we performed the proofs of the semantic relation in PVS. The mechanisation of synchronous parallel composition

is described in Section 6.7. A brief discussion of our mechanisation is presented in Section 6.9 which is then followed by a brief overview of related work.

6.2 PVS Datatypes

PVS has an expressive specification language augmented with classical higher order logic. The PVS specification language has a very sophisticated type system. The type system in PVS is even enriched by supporting uninterpreted type declarations, predicate subtypes, dependant types, enumerated types, and mechanisms for defining abstract datatypes such as lists, trees. Subtyping makes type checking more powerful. We briefly outline some datatypes that are used in mechanising the cCSP semantic models. Details of dataypes can be found in [91].

Uninterpreted Types

Uninterpreted type declarations introduce new types without specifying the elements of the type. The new type must be disjoint from all other types in the declaration. For example:

```
T  :  TYPE
```

declares an uninterpreted type `T`. The keyword `NONEMPTY_TYPE` declares an uninterpreted type with at least one element.

Predicated Subtypes

Subtype is a collection of elements from a given type. The elements in the subtype are characterised by a predicate on the given type which is called supertype. Given a type `T` and a predicate `p` on elements of `T`, a predicated subtype of `T` can be specified as

```
S : TYPE   =  {x: T | p(x) }
```

PVS recognises that any element of type `S` satisfies the predicates `p`. An equivalent, but compact declaration is:

```
S : TYPE = (p)
```

The parentheses surrounding `p` convert it to the corresponding subtype of `T`. An uninterpreted subtype of `T` can also be defined without explicitly expressing the predicated, as:

```
S : TYPEFROM T
```

Dependant Types

In dependant types, one type of component is dependant on other type. Function and tuples are permitted to contain dependant type components. For example,

```
T : TYPE = [x:  nat, {y:  nat | y <= x }]
```

declares a type `T`, whose elements are pair of natural numbers and second component is smaller than or equal to the first component.

Enumerated Types

Enumerated types are declared by listing explicitly the elements of the type. For example:

```
DAY : TYPE = { mon, tues, wed, thur, fri, sat, sun }
```

Constant Declarations

Constants can be declared for all objects that are declared in a PVS theory. Each constant has a specific type and as with types, they can be interpreted or uninterpreted. Declaring constants for uninterpreted types will generate a TCC which demands to prove its existence.

Recursive Definitions

PVS also supports recursive definitions. It is similar to constant declaration. A recursive definition has an explicit *measure* function, which ensures that the recursive definition terminates for its arguments. For example, consider

```
sum(n : nat) : RECURSIVE nat =
    IF n = 0 THEN 0 ELSE n + sum(n-1) ENDIF
MEASURE (LAMBDA n: n)
```

The value of the argument `n` is used as the measure and it must be decreased with each recursive call. A TCC will be generated to ensure this condition and PVS usually discharges the TCCs automatically.

Along with the above declarations, PVS also has variable and formula declarations. Formulas are constructed by using logical connectives such as `OR, AND, NOT, IMPLIES` etc and variables. The variables are bound by the quantifiers. The declaration of a formula associates a formula name with the logical formula by some keyword: `AXIOM, ASSUMPTION, LEMMA or THEOREM`. An `AXIOM` is not expected to have an associated proof whereas an `ASSUMPTION` is expected to be proved only by an importing theory. `LEMMA` or `THEOREM` always need to be proved.

Abstract Datatypes

One of the powerful features of the PVS specification language is its support for user-defined datatypes. The class of datatypes that can be defined in PVS is constrained [92] to ensure the resulting definitions and axioms are valid. A user defined datatype

is defined by using the keyword `DATATYPE` and PVS will generate valid axioms and definitions with respect to the given definition. Consider the definition of the list dataype below:

```
list[t : TYPE] : DATATYPE
BEGIN
    null : null?
    cons(car : t, cdr : list) : cons?
END list
```

Here list is declared as a type parameterised by `t` with two constructors `null` and `cons`. `null` takes no arguments and `cons` has two arguments where the first one is of type `t` and second argument is a `list` type. `null?` and `cons?` are recognisers (functions of type `[list -> bool]`) that are true for the empty list and the non-empty list respectively. The PVS typechecker enforces the following rules on the datatype declarations:

i) The constructors must be pairwise distinct.

ii) The recognisers must be pairwise distinct.

iii) The recursive occurrences of the datatype name must be positive to avoid cardinality contradictions.

Among the axioms generated automatically by PVS when the list datatype is type-checked, we are particulary interested in the following induction axiom:

```
list_induction : AXIOM
  FORALL (p : [list -> bool]) :
    p(null) AND
    FORALL (cons1_var : t), (cons2_var : list) :
       p(cons2_var) => p(cons(cons1_var, cons2_var))
  IMPLIES
    FORALL (list_var : list) : p(list_var)
```

The axiom is for a structural induction scheme which is applied automatically whenever an `INDUCT` command is applied to the theorem prover for a variable of list datatype.

Among the constant declarations generated by PVS, the definition that is significant for us is the `subterm`. It is defined on datatype objects and checks whether one objects occurs as a (not necessarily proper) subterm of another object. The proper subterm relation is defined by `<<`. The proper subterm relation is useful as a well-founded termination relation that can be given as the measure for a recursively defined function, for example, for the `list` datatype defined above the following definition can be generated:

```
<<(l1,l2 : list) : bool =
    CASES l2 OF
      null : FALSE,
      cons(cons1_var, cons2_var) : l1 = cons2_var OR
                                   l1 << cons2_var
    ENDCASES
```

The expression `CASES...ENDCASES` is a pattern matching constant declaration for abstract datatypes. In the above definition for `<<` the list argument `l2` is compared with the empty list pattern (`null`) or non-empty list pattern (`cons(cons1_var, cons2_var)`).

6.3 Embedding of cCSP in PVS

An embedding is a semantic encoding of one specification language into another, especially, to reuse the existing tools of the target language. It is possible to combine the complementary strengths of general-purpose verification systems and of the more methodical formal notations by a *semantic embedding* of the formal notation within the logic of the verification system. There are two main variants for the semantic embedding: *deep* and *shallow* embeddings [100]. In a deep embedding, the language and semantics of the method are fully formalised as an object in the logic of the specification language. In a shallow embedding, there is a syntactic translation of the objects of the method into semantically equivalent objects of the verification system. Shallow embedding concentrates on the semantic embedding of the guest logic into the host logic and it is easy to set up. We use PVS to embed the semantic models and their relationship and for our purpose we use shallow embedding in order to embed cCSP syntax and semantics into PVS.

The trace semantics of standard CSP has been mechanised in [30] and in [38, 40], where each follows a separate approach. In [30] Camilleri defined a dialect of CSP processes where each process has two components: a set of traces and an alphabet of events, whereas in [38, 40] processes are defined as sets of traces and traces are defined as lists of events. cCSP has a different semantic definitions than CSP and hence, our semantic definitions in PVS are different from their definitions. Besides, we not only define the trace semantics but also define the operational semantics of cCSP in PVS. The embedding of cCSP syntax along with its semantic models and the corresponding relationships are described in the following sections.

6.3.1 cCSP Syntax

First, we define the process terms to define the syntax of cCSP. PVS has its fixed syntax and it is not possible to use cCSP notations in PVS. Separate syntax is used

to define the standard and compensable processes. PVS supports overloading, hence the same notation can be used for both the operational and the trace semantics. In the operational semantics, process terms are used as processes. On the other hand, processes are denoted as a set of traces in the trace semantics.

TABLE 6.1: Syntax of standard process expressions

Operations	cCSP	PVS (operational)	PVS (trace)
SKIP	*SKIP*	Skip	SKIP
THROW	*THROW*	Throw	THROW
YIELD	*YIELD*	Yield	YIELD
Atomic Action	A	act(a)	act(a)
Choice	$P \square Q$	choice(P,Q)	choice(P,Q)
Sequential Composition	$P \; ; \; Q$	seq(P,Q)	seq(P,Q)
Interrupt Handler	$P \triangleright Q$	P |> Q	intr(P,Q)
Parallel Composition	$P \parallel Q$	para(P,Q)	parallel(P,Q)
Block operator	$[PP]$	blk(PP)	block(PP)

Table 6.1 denotes the notation that is used for standard processes.

The notations used for compensable processes are shown in Table 6.2. Operators are defined for both the operational and the trace semantics.

TABLE 6.2: Syntax of compensable process expressions

Operations	cCSP	PVS (operational)	PVS (trace)
SKIPP	*SKIPP*	Skipp	SKIPP
THROWW	*THROWW*	Throww	THROWW
YIELDD	*YIELDD*	Yieldd	YIELDD
Choice	$PP \square QQQ$	cchoice(PP,QQ)	cchoice(PP,QQ)
Sequential Composition	$PP \; ; \; QQ$	cseq(PP,QQ)	cseq(PP,QQ)
Compensation Pair	$P \div Q$	cpair(P,Q)	cpair(P,Q)
Parallel Composition	$PP \parallel QQ$	cpara(PP,QQ)	parallel(PP,QQ)

When mechanising a language in a theorem prover, one must ensure that any additional notation required in writing specifications is also mechanised along with its intended semantics. The PVS theorem prover has predefined libraries for various mathematical and logical theories which support most of the theoretical concepts that we need to mechanise cCSP. So we begin our account of mechanising cCSP using these built-in facilities to define the notion of process terms, events, processes and traces.

6.3.2   Events, Traces and Processes

There are two types of events in cCSP: observable (normal)[1] and terminal. Observable events cause the transition of a process from one state to another whereas terminal events cause the termination of a process. These two types of events are defined in PVS and then the terminal events are defined as constants of terminal type as follows:

```
normal   : TYPE
terminal : TYPE+
tick, yield, throw, bottom : terminal
```

The keyword `TYPE+` indicates that `terminal` is a *non-empty* type, that allows to define constants of that type.

The traces of cCSP, defined in an earlier chapter, have a sequence of observable events (Σ^*) followed by a terminal event $(\omega \in \Omega)$. Following this definition of traces, we define traces of standard processes in PVS as a pair consisting of a list of normal events and a terminal event. This definition ensures that traces are non-empty (at least there is a terminal event). As compensable processes have forward and compensation traces, these traces are defined as a pair of standard traces.

```
trace      : TYPE = [list[normal],terminal]
comp_trace : TYPE = [trace, trace]
```

We define a property for traces to add an event to a trace. Adding an event to a trace is adding it in front of the trace. Later, we use this property in the inductive proofs for each term of the language: `add2trace(a:normal,t:trace) : trace = (cons(a,t'1),t'2)`

Processes are defined as a set of traces and all the general rules about sets immediately apply to processes. Standard processes are defined as sets of traces and compensable processes are defined as sets of compensable traces (can also be defined as sets of pair of traces).

```
process      : TYPE = setof[trace]
comp_process : TYPE = setof[comp_trace]
```

6.3.3   Process-Algebra terms

The proofs about properties of process algebra often use induction on the structure of terms of the algebra. We also apply induction on process terms for the correspondence proofs of the semantic models. PVS generates an induction scheme for abstract

---

[1]we use observable and normal interchangeably

datatypes and hence, it is convenient to model process terms as an abstract datatype. PVS has a mechanism for user-defined (recursive) datatypes. PVS provides mechanism to define abstract datatypes of certain class which includes all of the tree-like recursive data structures that are freely generated by a number of constructor operations. For example, the abstract datatype of lists is generated by the constructor `null` and `cons` and similarly, the abstract datatype of stack is generated by the constructors `empty` and `push`. PVS excludes datatypes such as unordered list, bags etc. that are not freely generated. For example, two different sequence of insertions of elements into a bag can yield equivalent bags. A detailed discussion of PVS abstract datatypes can be found in [92].

In cCSP, we have terms for standard and compensable processes. An important point is that the terms are mutually dependant on each other. The process term inside a block operator is a compensable process, but the block operator represents a standard process and the proofs concerned with the block operator depend on the proofs of compensable processes. Similarly, the compensation pair operator represents a compensable process, but consists of two standard processes and proofs are dependent on the proofs for standard processes. Therefore, we have to use mutually recursive datatypes to define the process terms.

Although mutually recursive datatypes arise quite frequently in language and specifications, they are not directly admissible by PVS. However, with the extended support of *sub-datatype* [92, 110], we can use the two mutually recursive datatypes as a single datatype. A sub-datatype collects together groups of constructors of a dataype that form one part of a mutually recursive datatype definition. The process algebra terms with two sub-datatypes are defined in PVS as follows:

```
pa_terms  : DATATYPE WITH SUBTYPES stand, comp
  BEGIN
      Skip    : skip?   : stand
      Throw   : throw?  : stand
      Yield   : yield?  : stand
      Skipp   : skipp?  : comp
      Throww  : throww? : comp
      Yieldd  : yieldd? : comp
      act(a:normal)              : act?      : stand
      seq(P1:stand, Q1:stand)    : seq?      : stand
      cseq(P : comp, Q : comp)   : c_seq?    : comp
      para(P1:stand, Q1:stand)   : para?     : stand
      cpara(P : comp, Q :comp)   : c_para    : comp
      choice(P1: stand, Q1: stand) : choice?   : stand
      cchoice(P : comp, Q : comp)  : c_choice? : comp
      |>(P1: stand, Q1: stand)   : inthnd?   : stand
      cpair(P1: stand, Q1 : stand) : cpair?    : comp
      blk(P : comp)              : blk?      : stand
      ax(P:comp,P1:stand)        : ax?       : comp
      nul                        : nul?      : stand
  END pa_terms
```

We define a single datatype `pa_terms` which consists of two sub-datatypes: 'stand' for standard processes and 'comp' for compensable processes. We can now define process terms of type `stand` and `comp`. We define the process term `nul` to denote the null (0) process that we have used in defining the operational semantics of the process terms. The process term `ax` is defined here to denote the auxiliary construct which is introduced during defining the operational semantics of sequential composition of compensable processes (discussed in Chapter 3) and it is used in the inductive case in the proof of the lemma for compensable sequential composition.

When this definition is type checked in PVS, a new file is generated containing a large number of useful definitions and properties of the datatype. An induction scheme is also generated expressing that a property p on terms can be proved by showing that it holds for all atoms and by proving that it holds for all the operators if it holds for the subterms.

The semantic models of cCSP are defined in the following sections. The operators are defined in terms of their semantic models and laws of these operators can be proved from these semantic definitions. We avoided the approach of directly encoding laws as axioms as it would allow us to introduce inconsistencies in the logic.

6.4 Mechanising the Trace Semantics

In PVS, keeping the same fashion as in the original semantic definitions, operators are defined initially at trace level and then lifted to sets of traces to define processes. Each operator is defined separately. The trace semantics for standard and compensable processes are defined in the following sections.

6.4.1 Standard Processes

Atomic Action

The atomic action can perform an atomic event and the terminates. The trace semantics is defined as follows:

```
act(a:normal) : process = {t:trace | t = (cons(a,null),tick)}
```

Basic Processes

The standard basic processes of cCSP only have terminal events in their traces. The traces of basic processes are defined as follows:

```
SKIP  : process = {t: trace | t = (null,tick)}
THROW : process = {t: trace | t = (null,throw)}
YIELD : process = {t: trace | t = (null, yield)
                       OR  t = (null, tick)}
```

Choice

The traces of the process in the choice operator is the union of traces from the individual processes from the choice:

```
choice(P,Q:process): process = union(P,Q)
```

Sequential composition

For traces p and q, their sequential composition $(p \; ; q)$ is defined in such a way that if trace p ends with a $\checkmark$, the trace q will be augmented with the observable events of p and $\checkmark$ will be hidden from the environment. If p ends with a terminal event other than a $\checkmark$ then the sequential composition of the two traces only returns the events of p and the events of q are discarded. The trace definition in PVS is given as follows:

```
seq(p,q:trace) : trace =
  IF PROJ_2(p) = tick THEN
      (append(PROJ_1(p),PROJ_1(q)),PROJ_2(q))
  ELSE p
  ENDIF
```

`PROJ_i` is the ith element of a tuple. The ith element from a tuple `t` can also be represented by `t'i`. We use these notations interchangeably. The sequential composition of processes is defined as follows:

```
seq(P,Q): process = {t: trace | EXISTS(p:(P),q:(Q)): t = seq(p,q)}
```

Interrupt Handler

The trace definition of the interrupt handler is dual to the definition of sequential composition where the traces of the second process are augmented when the first process terminates with a `throw` rather than a `tick`. The definition is given below:

```
intr(p,q:trace) : trace =
  IF PROJ_2(p) = throw THEN
     (append(PROJ_1(p),PROJ_1(q)),PROJ_2(q))
  ELSE p
  ENDIF

intr(P,Q:process) : process =
 {t : trace | EXISTS (p:(P),q:(Q)): t = intr(p,q) }
```

Parallel Composition

We first define the asynchronous composition of concurrent processes, where processes interleave over observable events and synchronise over terminal events. Later, we will extend the asynchronous definitions in order to define synchronous processes.

The interleaving of the list of observable events is defined in PVS as follows:

```
interleave(t1,t2,t:list[normal]): RECURSIVE bool =
  CASES t OF
      null: null?(t1) AND null?(t2),
      cons(x,y):
      (cons?(t1) AND car(t1)= x AND interleave(cdr(t1),t2,y))
       OR (cons?(t2) AND car(t2)= x AND interleave(t1,cdr(t2),y))
  ENDCASES
  MEASURE length(t)
```

`interleave(t1,t2,t)` holds when `t` is a valid interleaving of `t1` and `t2`.

PVS allows a restrictive form of recursive definition. Mutual recursion is not allowed and the function must be total, so that the function is defined for every value of its domain. In order to ensure this, a `MEASURE` function is required. The function has the same domain as the definition, but its range is `nat` (in the above case) or ordinals (will be used later). The `MEASURE` function is defined to show that the definition terminates, by generating an obligation that the `MEASURE` decreases with each call ensuring that the definition is well-founded.

In the asynchronous version of the parallel operator, we do not need to consider the `bottom` event. The synchronisation of the terminal events is defined as follows:

```
w,w1,w2: VAR terminal
parallel(w)(w1,w2): bool =
 IF w = throw THEN
           w1 = throw AND w2 = throw
```

```
            OR w1 = throw AND w2 = yield
            OR w1 = throw AND w2 = tick
            OR w1 = yield AND w2 = throw
            OR w1 = tick  AND w2 = throw
    ELSIF w = yield THEN
              w1 = yield AND w2 = yield
            OR w1 = yield AND w2 = tick
            OR w1 = tick  AND w2 = yield
    ELSE      w1 = tick  AND w2 = tick
    ENDIF
```

Finally, the parallel composition of traces is defined by combining the interleaving of list of event and synchronisation of the terminal events. In PVS, the definition is given as follows:

```
parallel(r:trace)(p,q: trace): bool =
  interleave(proj_1(p),proj_1(q),proj_1(r))
  AND parallel(proj_2(r))(proj_2(p),proj_2(q))
```

After defining parallel composition at the trace level, we can then define the parallel composition of two processes.

```
parallel(P,Q: process): process =
  {t:trace | EXISTS (p:(P),q:(Q)): parallel(t)(p,q)}
```

6.4.2 Compensable Processes

Compensable processes have both forward as well as compensation behaviour. While mechanising compensable processes we mechanise both forward and compensation traces. Traces for compensable processes are defined in such a way that the compensations attached to the processes will be augmented in the proper order after termination of the forward behaviour.

Choice

The choice operator for compensable processes is defined in the same way as for standard processes:

```
cchoice(PP,QQ:comp_process): comp_process = union(PP,QQ)
```

Sequential Composition

The sequential composition for compensable processes is defined in such a way that after termination of the forward behaviour of the composition, the attached compensation will be augmented in reverse order to that of the forward behaviour. For traces *pp* and *qq*, their sequential composition (*pp* ; *qq*) is defined in such a way that when the forward behaviour of *pp* terminates with a ✓, the forward behaviour of *qq* is augmented with the observable forward behaviour of *pp* and the compensations from *pp* and *qq* (*p* and *q*) will be composed in reverse order. But if *pp* terminates with a terminal event other than a ✓, the traces *qq* will not be considered at all and only compensation from *pp* is kept for future reference. The mechanisation in PVS is defined as follows:

```
seq(pp,qq:comp_trace) : comp_trace =
    IF (proj_2(proj_1(pp)) = tick ) THEN
    (seq(proj_1(pp),proj_1(qq)),seq(proj_2(qq),proj_2(pp)))
    ELSE pp
    ENDIF
```

The composition of process is defined by lifting the sequential composition on traces to sets of traces as follows:

```
  seq(PP,QQ : comp_process) : comp_process =
    {tt:comp_trace | EXISTS (pp:(PP),qq:(QQ)): tt = seq(pp,qq)}
```

Compensation Pair

The compensation pair is defined from two standard processes: one is a forward process and another is the compensation. The trace semantics is defined in such a way that if the terminal event of the forward trace is a tick event then the compensation trace is augmented to the forward traces. For other terminal events traces of the compensation are discarded. The definition of traces and processes in PVS is given as follows:

```
  pair(p,q:trace): comp_trace =
    IF proj_2(p) = tick THEN
     (p,q)
    ELSE
     (p,(null,tick))
    ENDIF

  pair(P,Q:process): comp_process =
    {tt: comp_trace | EXISTS (p:(P),q:(Q)):
            tt = pair(p,q)}
```

Parallel Composition

The traces of a parallel composition of compensable processes are defined in a similar way to standard processes, the only difference is that after the termination of the forward processes, the corresponding compensations are accumulated in parallel. So, by using the parallel composition defined for standard processes, the traces here are defined as the parallel composition of forward behaviour, which then follows the parallel composition of the compensations. Both the trace and process definition of compensable parallel composition are defined in PVS as follows:

```
parallel(rr:comp_trace)(pp,qq : comp_trace): bool =
 interleave(proj_1(proj_1(pp)),proj_1(proj_1(qq)),proj_1(proj_1(rr))) AND
 parallel(proj_2(proj_1(rr)))(proj_2(proj_1(pp)),proj_2(proj_1(qq)))  AND
 interleave(proj_1(proj_2(pp)),proj_1(proj_2(qq)),proj_1(proj_2(rr))) AND
 parallel(proj_2(proj_2(rr)))(proj_2(proj_2(pp)),proj_2(proj_2(qq)))

 parallel(PP,QQ : comp_process): comp_process =
    { tt : comp_trace | EXISTS (pp:(PP),qq:(QQ)) :
      parallel(tt)(pp,qq)  }
```

Transaction Block

A transaction block is a standard process, but its semantics requires an understanding of the semantics of compensable processes. So, the definition is placed here with compensable processes. Recall that, a transaction block will be represented by traces instead of pairs of traces. The semantics of the block specifies that if the forward part of the process inside the block terminates with a ✓, then the trace of the block is the trace of the forward behaviour and if the terminal event is a throw (!), then the behaviour of the compensation is augmented to the observable behaviour of the forward behaviour.

```
  block(pp:comp_trace) : trace =
   IF (pp'1)'2 = throw THEN
   (append((pp'1)'1,(pp'2)'1),(pp'2)'2)
   ELSE  pp'1
   ENDIF

 block(PP:comp_process): process =
   { t: trace | EXISTS (pp:(PP)): t = block(pp) }
```

In the next section, we describe the mechanisation of the operational semantics of standard and compensable processes.

6.5   Mechanising the Operational Semantics

The operational semantics is defined by using labelled transitions of the form $P \xrightarrow{e} P'$ where the event e makes the transition of a process from state P to P' and the transition is labelled by e. We have discussed in Chapter 3 that there are two types of transitions in our semantic definition: transitions by normal events and transitions by terminal events. For example, these two transitions are defined for standard processes as follows:

$$P \xrightarrow{a} P' \qquad (a \in \Sigma)$$
$$P \xrightarrow{\omega} 0 \qquad (\omega \in \Omega)$$

While defining the transition relations in PVS, we follow the same approach and define the normal and terminal transitions separately. The transition relations are defined in such a way that they return a boolean value determining whether there is a transition between two states of a process. We define two types of transitions in PVS for each operator.

Recall that in Chapter 4, separate equations are derived for normal and terminal transition relations from the transition rules of each operator in order to support the correspondence proofs. We use those derived equations (e.g. equations 4.3, 4.4, page 49) to define the normal and terminal transitions for process terms in PVS.

First, we define the terminal transitions for process terms. Recall that the terminal transitions for standard processes are different from those of compensable processes (Chapter 3). We define the terminal transitions for both standard and compensable processes together in PVS. The null process was introduced in Chapter 3 to define the transition rules. We define a standard process term `nul` to represent the null process in the definition of process algebra terms. This definition allows us to combine the definition of terminal transitions of both standard and compensable processes. The terminal transition for the process terms are defined in PVS as follows:

```
wtrans(w: terminal)(P:pa_terms,P1:stand): RECURSIVE bool =
CASES P OF
     Skip      : w = tick  AND P1 = nul,
     Throw     : w = throw AND P1 = nul,
     Yield     : w = yield AND P1 = nul
                 OR  w = tick AND P1 = nul,
    choice(Q,R): wtrans(w)(Q,nul) AND P1 = nul
                 OR wtrans(w)(R,nul) AND P1 = nul,
    seq(Q,R)   : wtrans(tick)(Q,nul) AND wtrans(w)(R,nul) AND P1 = nul
                 OR
                 w /= tick AND wtrans(w)(Q,nul) AND P1 = nul,
```

```
    |>(Q,R)    : wtrans(throw)(Q,nul) AND wtrans(w)(R,nul) AND P1 = nul
                 OR
                 w /= throw AND wtrans(w)(Q,nul) AND P1 = nul,
    para(Q,R)  : EXISTS (w1,w2) :
                 wtrans(w1)(Q,nul) AND wtrans(w2)(R,nul) AND
                 parallel(w)(w1,w2) AND P1 = nul,


 cchoice(QQ,RR): EXISTS (Q:stand): wtrans(w)(QQ,Q) AND P1 = Q
                 OR
                 EXISTS (R:stand):wtrans(w)(RR,R) AND P1 = R,
   cseq(QQ,RR): EXISTS (Q,R : stand):
                 wtrans(tick)(QQ,Q) AND wtrans(w)(RR,R) AND
                 P1 = seq(R,Q)
                 OR
                 wtrans(w)(QQ,P1) AND w /= tick,
  cpara(QQ,RR): EXISTS (w1,w2:terminal,Q1,R1:stand):
                 wtrans(w1)(QQ,Q1) AND wtrans(w2)(RR,R1) AND
                 parallel(w)(w1,w2) AND P1 = para(Q1,R1),
    cpair(Q,R): wtrans(tick)(Q,nul) AND P1= R
                 OR
                 wtrans(w)(Q,nul) AND
                 w/= tick AND P1 = Skip,
       blk(QQ): (EXISTS (Q1:stand) :
                 wtrans(throw)(QQ,Q1) AND
                 wtrans(w)(Q1,nul) AND P1 = nul)
                 OR
                 (EXISTS (Q:stand):
                 w = tick AND wtrans(w)(QQ,Q) AND P1 = nul),
     ax(QQ,R): EXISTS Q: wtrans(w)(QQ,Q) AND
                 P1 = seq(R,Q)
  ELSE FALSE
  ENDCASES
 MEASURE P BY <<
```

Note that the PVS MEASURE keyword introduces the measure used to prove the well-foundedness of the recursion and thus termination of the function. Here the termination of wtrans is guaranteed since P is decreasing w.r.t. the recursive subterm ordering << in the recursive calls.

The transitions by normal events for both standard and compensable processes are defined as follows:

```
ntrans(a:normal)(Pa:pa_terms,Pa1:pa_terms): RECURSIVE bool =
  CASES Pa OF
   act(a)        : Pa1 = Skip,
   choice(Q,R)   : ntrans(a)(Q,Pa1) OR ntrans(a)(R,Pa1),
   seq(Q,R)      : EXISTS Q1 :  ntrans(a)(Q,Q1)  AND
                   Pa1 = seq(Q1,R)
                   OR wtrans(tick)(Q,nul)       AND
                   ntrans(a)(R,Pa1),
   |>(Q,R)       : EXISTS Q1 : ntrans(a)(Q,Q1)   AND
                   Pa1 = |>(Q1,R)
                   OR wtrans(throw)(Q,nul)       AND
                   ntrans(a)(R,Pa1),
   para(Q,R)     : EXISTS Q1: ntrans(a)(Q,Q1)    AND
                   Pa1 = para(Q1,R)
                   OR EXISTS R1: ntrans(a)(R,R1) AND
                   Pa1 = para(Q,R1),

   cchoice(QQ,RR)  : ntrans(a)(QQ,Pa1) OR ntrans(a)(RR,Pa1),
   cpara(QQ,RR)    : EXISTS (QQ1:comp): ntrans(a)(QQ,QQ1) AND
                      Pa1 = cpara(QQ1,RR)
                      OR
                      EXISTS (RR1:comp): ntrans(a)(RR,RR1) AND
                      Pa1 = cpara(QQ,RR1),
   cseq(QQ,RR)     : EXISTS (QQ1:comp): ntrans(a)(QQ,QQ1) AND
                      Pa1 = cseq(QQ1,RR)
                      OR EXISTS (RR1:comp,Q:stand):
                      wtrans(tick)(QQ,Q)          AND
                      ntrans(a)(RR,RR1)           AND
                      Pa1 = ax(RR1,Q),
   cpair(Q,R)      : EXISTS (Q1:stand): ntrans(a)(Q,Q1)   AND
                      Pa1 = cpair(Q1,R),
   ax(QQ,R)        : EXISTS (QQ1:comp): ntrans(a)(QQ,QQ1) AND
                      Pa1 = ax(QQ1,R),
   blk(QQ)         : EXISTS (QQ1:comp): ntrans(a)(QQ,QQ1) AND
                      Pa1 = blk(QQ1)
                      OR
                      EXISTS (Q:stand): wtrans(throw)(QQ,Q)      AND
                      ntrans(a)(Q,Pa1)
      ELSE FALSE
    ENDCASES
  MEASURE Pa BY <<
```

Standard and compensable processes have similar transitions for normal events. Recall that the basic processes do not have any normal transitions.

6.6 Mechanising Semantic Relationships

Having the mechanisation of the semantic models, in this section, we outline in PVS the proof of the relationships between the semantic models. We follow similar steps in deriving the correspondence as in the hand proofs and hence provide the required definitions.

Proving the lemmas defined in Chapter 4 for each operator is the key factor to proving the correspondence between the semantic models. The section shows how we defined all the lemmas and proved them by applying induction. These lemmas establish the relationship by proving the correspondence between the derived traces from the operational semantics and the originally defined traces. First, we define how to derive traces from operational rules in PVS. A derived trace is defined as a transition of a process term by a trace. It consists of a transition by a list of normal events followed by a terminal transition. `trans_list` defines the transition by a list of normal events:

```
trans_list(s:list[normal])(P,P1): RECURSIVE bool =
   CASES s OF
    null : P = P1,
    cons(h,tail): EXISTS (Q:stand): ntrans(h)(P,Q) AND
                 trans_list(tail)(Q,P1)
   ENDCASES
 MEASURE length(s)
```

Standard and compensable process terms have different types of transitions. We define them separately in the following sections.

6.6.1 Standard Processes

For standard processes, the transition of a process term by a trace is defined by combining `trans_list` with the terminal transition `wtrans`. The transition of a process term by a trace always ends with a null process (0). The transition of a process by a trace is defined as follows:

```
trans_trace(t:trace)(P,N:stand) : bool =
 EXISTS (Q:stand) :
   trans_list(t'1)(P,Q) AND wtrans(t'2)(Q,N) AND N = nul
```

In the basic steps of the inductive proofs of the lemmas defined in earlier chapters, there is a transition which shows that a transition of a process term by a trace containing only one terminal event is the same as a terminal transition, denoted as follows:

$$P \xrightarrow{\langle \omega \rangle} 0 \; = \; P \xrightarrow{\omega} 0$$

In order to support the inductive proofs in PVS, we define it in PVS as a lemma, shown as follows:

```
nul_eq : LEMMA
  FORALL P :  trans_trace((null,w))(P,nul) = wtrans(w)(P,nul)
```

We also embed the Theorem 4.4 (page 47) concerning the evolution of a standard process term to a null process. This theorem is required in the inductive proofs of some lemmas:

```
th1 : LEMMA
  FORALL P: EXISTS (s,w): trans_trace(s,w)(P,nul)
```

By using the definition of transition by a trace and from the original trace definitions we state the required lemmas for each operator in the following sections.

Sequential Composition

The correspondence proof between the derived traces and the originally defined traces of a sequential composition is performed by proving the Lemma 4.6 defined in Chapter 4. The corresponding PVS definition is given as follows:

```
s,s1,s2 : VAR list[normal]
w,w1,w2 : VAR terminal
seq_lemma : LEMMA
  trans_trace((s,w))(seq(P,Q),nul) =
   EXISTS (s1,w1,s2,w2) :
       (s,w) = seq((s1,w1),(s2,w2)) AND
       trans_trace((s1,w1))(P,nul)  AND
       trans_trace((s2,w2))(Q,nul)
```

In the lemma, traces are explicitly defined as pairs to make it easy to apply the induction scheme. `seq_lemma` defined above has been proved by induction over `s` which gives the required induction scheme that we want to use on traces. While proving the base case of the induction there are two cases based on whether or not `P` terminates with a ✓. Suitable instantiation will prove the first cases.

Parallel Composition

The lemma for the asynchronous parallel composition (Lemma 4.10, page 56) is defined in PVS in a similar fashion as sequential composition:

```
para_lemma : LEMMA
   trans_trace((s,w))(para(P,Q),nul) =
   EXISTS (s1,w1,s2,w2) :
     parallel((s,w))((s1,w1),(s2,w2)) AND
     trans_trace((s1,w1))(P,nul)      AND
     trans_trace((s2,w2))(Q,nul)
```

Similar to sequential composition, we prove the lemma by applying induction over `s`.

6.6.2 Compensable Processes

Compensable processes have forward and compensation behaviour. The compensation behaviour is the same as that of standard processes. We have shown in an earlier chapter how to reuse the proofs from standard processes for compensation. Therefore, we only need to derive traces from the forward behaviour and show their correspondence with the original definition of forward traces.

As for standard processes, we define the transition of compensable process terms by a list of normal events as 'ctrans_list'. By combining it with the terminal transition, we define the transition of a compensable process term by the forward traces as follows:

```
ftrans_trace(t:trace)(PP:comp,P:stand) : bool =
  EXISTS QQ : ctrans_list(t'1)(PP,QQ) AND wtrans(t'2)(QQ,P)
```

We define the transition of a compensable process term by combining the transition by the forward traces and the transition by the compensation traces (same as standard trace transition) in PVS as follows:

```
ctrans_trace(tt:comp_trace)(PP:comp,N:stand): bool =
 EXISTS P: ftrans_trace(tt'1)(PP,P) AND
          trans_trace(tt'2)(P,N) AND N = nul
```

We also extend the definitions of 'nul_eq' and 'th1' to reflect the changes for compensable processes:

```
  c_null_eq: LEMMA
   FORALL PP: ftrans_trace((null,w))(PP,P) = wtrans(w)(PP,P)
  c_th1: LEMMA
     FORALL PP: EXISTS P,s,w: ftrans_trace((s,w))(PP,P)
```

Sequential Composition

For sequential composition of compensable processes Lemma 4.7 (page 52) is defined for
the lifted forward behaviour:

$$
\begin{aligned}
(PP \; ; QQ) \xrightarrow{t} R \;\; &= \;\; \exists P, Q, p, q \cdot t = (p \; ; \; q) \wedge PP \xrightarrow{p} P \wedge QQ \xrightarrow{q} Q \\
&\wedge \;\; R = COND\,(last(p) = \checkmark, (Q \; ; \; P), \, P)
\end{aligned}
$$

Only the definition of the forward trace transition is taken into account to define the
above lemma. PVS has support for defining the *COND* expression in the lemma, which
has the same definition that we required here. The lemma is defined in PVS as follows:

```
        cseq_lemma : LEMMA
            ftrans_trace((s,w))(cseq(PP,QQ),R) =
            EXISTS P,Q, s1,w1,s2,w2 :
                 (s,w) = seq((s1,w1),(s2,w2)) AND
                 ftrans_trace((s1,w1))(PP,P)   AND
                 ftrans_trace((s2,w2))(QQ,Q)   AND
                 R = COND
                         tick?(w1) -> seq(Q,P),
                         ELSE -> P
                     ENDCOND
```

Parallel Composition

For the lifted forward traces for parallel composition of compensable process terms,
Lemma 4.11 is defined in PVS as follows:

```
        cpara_lemma : LEMMA
            ftrans_trace((s,w))(cpara(PP,QQ),R) =
            EXISTS (P,Q,s1,s2,w1,w2) :
                    parallel((s,w))((s1,w1),(s2,w2)) AND
                    ftrans_trace((s1,w1))(PP,P)        AND
                    ftrans_trace((s2,w2))(QQ,Q)        AND
                    R = para(P,Q)
```

Compensation Pair

The compensation pair is the basic way of constructing a compensable process. A compensation pair $(P \div Q)$ consists of two standard processes: forward behaviour (P) and compensation (Q). The Lemma 4.12 (page 59) is defined to show the correspondence for a compensation pair:

$$(P \div Q) \xrightarrow{(t,t')} 0 \;=\; \exists\, p, q \cdot (t, t') = (p \div q) \,\wedge\, P \xrightarrow{p} 0 \,\wedge\, Q \xrightarrow{q} 0$$

In order to define this lemma in PVS, we not only need the forward traces but also the compensation traces. We define the lemma by using the definition `ctrans_trace`. The PVS definition is given as follows:

```
pair_lema : LEMMA
        ctrans_trace((s,w),(s3,w3))(cpair(P,Q),nul) =
        EXISTS (s1,w1,s2,w2) :
                    ((s,w),(s3,w3)) = pair((s1,w1),(s2,w2)) AND
                    trans_trace((s1,w1))(P,nul) AND
                    trans_trace((s2,w2))(Q,nul)
```

We apply induction over `s`. According to the semantics of the compensation pair, the trace from the compensation can be either empty or a trace from `Q`. Unlike other operators, here we need to consider both forward and compensation behaviour. In order to support the proof, it is also required to use both trace and transition rules. We need to use a transition rule in the base case of the proof and in order to support the proof we define as axiom from the derived equation 4.22 (page 59). As it is a direct consequence of the transition rule, we define it as an axiom.

```
nul_induct : AXIOM
 ctrans_trace((null,w),(s3,w3))(cpair(P,Q),nul) =
  (w = tick AND wtrans(tick)(P,nul) AND trans_trace(s3,w3)(Q,nul))
  OR
  w /= tick AND wtrans(w)(P,nul) AND Q = Skip AND
  null?(s3) AND w3 = tick AND wtrans(tick)(Skip,nul)
```

In the inductive case of the proof, we use a supporting lemma 'add2pair' which states that adding an event to the traces of the pair is adding it in front of the forward trace of the pair:

```
add2ctrace(a:normal,t3:comp_trace) :
    comp_trace = ((cons(a,(t3'1)'1),(t3'1)'2),t3'2)


add2pair : LEMMA
    add2ctrace(a,pair(p,q)) = pair((cons(a,p'1),p'2),q)
```

6.6.3 Transaction Block

The important feature of the transaction block is that it returns the behaviour of a standard process from the behaviour of the compensable process inside the block. The rules for compensable processes are applied for the process inside the block.

Lemma 4.13 (page 60) is defined to derive the correspondence for the block operator:

$$[PP] \xrightarrow{t} 0 \;=\; \exists\, p, p' \cdot t = [p, p'] \,\wedge\, PP \xrightarrow{p,p'} 0$$

The lemma is defined in PVS as follows:

```
 block_lemma : LEMMA
         trans_trace((s,w))(blk(PP),nul) =
         EXISTS (s1,w1,s2,w2):
                 (s,w) = block((s1,w1),(s2,w2)) AND
                 ctrans_trace((s1,w1),(s2,w2))(PP,nul)
```

6.7 Mechanising Synchronous Semantic Models

We have extended the semantic models to define the synchronisation between observable events for the parallel operator in Chapter 5. We have already shown the mechanisation

of the asynchronous version of the parallel processes. This section is devoted to the mechanisation of the semantics and their relationships for the synchronous concurrency operator.

Before mechanising the semantic models of the operator, first we define the syntax for the operator for both the trace and the operational semantics. The syntax defined in Table 6.1 and in Table 6.2 are extended with the synchronisation operator. To denote the trace semantics, we write `full_parallel(X)(P,Q)` for standard processes `cfull_parallel(X)(PP,QQ)` for compensable processes. Here X represents the synchronisation set. The process algebra terms are also extended to include the synchronising operator for both standard and compensable processes:

```
pa_terms : DATATYPE WITH SUBTYPES stand, comp
   BEGIN
      .....
      synpara(X:setof[normal], P1:stand, Q1:stand) : synpara?  :stand
      csynpara(X:setof[normal], P:comp, Q:comp)    : csynpara? :comp
            .....
   END pa_terms
```

6.7.1 Trace Semantics (Synchronous Parallel)

The trace semantics of the asynchronous processes is extended to support synchronisation. In the synchronisation of processes, some events will synchronise and others will interleave. In the following sections, we define the trace semantics for both standard and compensable processes.

Standard Processes

In order to define the synchronous composition of processes we need to define both synchronising and interleaving events. So, we can extend the asynchronous definition to consider all the scenarios of synchronised composition. Note that while mechanising asynchronous semantics, the interleaving of observable events and synchronisation of terminal events are defined separately in PVS and later they are put together in the trace definition. However, this kind of separation is not possible in the synchronous definition as there is now synchronisation between observable and terminal events which will result in either interleaving of events or a bottom event, e.g., $a \& \bot = \bot$. The synchronisation of events, e.g., $a \& a = a$, $a \& a' = \bot$, $a \& \bot = \bot$ is encoded directly in trace composition.

The synchronisation between terminal events is also extended to include the bottom event in the definition. Instead of modifying the existing definition we define a new

synchronisation where the previous definition is used in the appropriate case. The extended definition is give below:

```
syn_parallel(w3:terminal)(w1,w2:terminal) : bool =
    IF w3 = bottom THEN
        w1 = bottom OR w2 = bottom
      ELSE parallel(w3)(w1,w2) ENDIF
```

We let X be the set of synchronising normal events. The trace semantics of the synchronised parallel operator (Definition 5.1, page 69) is defined in PVS as follows:

```
full_parallel(X)((s1,w1))((s2,w2))((s3,w3)) : RECURSIVE bool =
 CASES s3 OF
  null:
      null?(s1) AND null?(s2)  AND syn_parallel(w3)(w1,w2)
   OR cons?(s1) AND X(car(s1)) AND null?(s2) AND w3 = bottom
   OR cons?(s2) AND X(car(s2)) AND null?(s1) AND w3 = bottom
   OR cons?(s1) AND X(car(s1)) AND cons?(s2) AND X(car(s2)) AND
      car(s1) /= car(s2) AND w3 = bottom,
 cons(a,tail):
      IF X(a) THEN
        cons?(s1) AND cons?(s2)     AND
        car(s1) = a AND car(s2) = a AND
        full_parallel(X)((cdr(s1),w1))((cdr(s2),w2))((tail,w3))
      ELSE
        cons?(s1) AND car(s1) = a AND
        full_parallel(X)((cdr(s1),w1))((s2,w2))((tail,w3))
     OR cons?(s2) AND car(s2) = a AND
        full_parallel(X)((s1,w1))((cdr(s2),w2))((tail,w3))
      ENDIF
  ENDCASES
 MEASURE length(s3)
```

According to the PVS style for sets, here X acts as a boolean function on normal events.

The trace definition is lifted to the set of traces to define the synchronisation of processes. Synchronisation of processes are defined as follows:

```
 full_parallel(X)(P,Q : process): process =
{t : trace | EXISTS (p:(P),q:(Q),s1,w1,s2,w2,s3,w3):
    p = (s1,w1) AND q = (s2,w2) AND t = (s3,w3) AND
   full_parallel(X)((s1,w1))((s2,w2))((s3,w3))
  }
```

Compensable Processes

Compensable processes are defined in a similar way to standard processes with the compensation behaviour added. The trace semantics of compensable processes (Definition 5.3 and 5.4, page 70) is defined in PVS as follows:

```
cfull_parallel(X)((p,p1))((q,q1))((r,r1)) : bool =
      (full_parallel(X)(p)(q)(r) AND
      full_parallel(X)(p1)(q1)(r1) AND
      r'2 /= bottom)
  OR  full_parallel(X)(p)(q)(r) AND
      r'2 = bottom AND null?(r1'1) AND
      r1'2 = bottom

cfull_parallel(X)(PP,QQ:comp_process): comp_process =
{ tt:comp_trace | EXISTS (pp:(PP),qq:(QQ)) :
                  cfull_parallel(X)(pp)(qq)(tt)
}
```

6.7.2 Operational Semantics (Synchronous Parallel)

The transition rules for asynchronous composition are extended to define synchronous composition.

In a normal transition, processes either synchronise or interleave with each other. We let X be the set of synchronising events. By extending the asynchronous transition rules, we define the synchronised transition rules as follows:

```
synpara(X,Q,R):
    IF X(a) THEN
      EXISTS Q1,R1 : ntrans(a)(Q,Q1) AND  ntrans(a)(R,R1) AND
                     Pa1 = synpara(X,Q1,R1)
    ELSE
     EXISTS Q1     : ntrans(a)(Q,Q1)  AND
                     Pa1 = synpara(X,Q1,R)
  OR EXISTS R1     : ntrans(a)(R,R1)   AND
                     Pa1 = synpara(X,Q,R1)
    ENDIF
    csynpara(X,QQ,RR) :
      IF X(a) THEN
        EXISTS QQ1,RR1 : ntrans(a)(QQ,QQ1) AND ntrans(a)(RR,RR1) AND
                         Pa1 = csynpara(X,QQ1,RR1)
      ELSE
```

```
    EXISTS QQ1        : ntrans(a)(QQ,QQ1)         AND
                        Pa1 = csynpara(X,QQ1,RR)
    OR EXISTS RR1     : ntrans(a)(RR,RR1)         AND
                        Pa1 = csynpara(X,QQ,RR1)
```

In Section 5.4.1, and 5.4.2, we have defined transition rules for both standard, and compensable processes. We have defined rules for both cases, where synchronising processes synchronise with each other, and synchronising processes introduce a bottom when they fail to synchronise over synchronising events. All these transition rules are essential for a case by case analysis for the lemmas of the synchronising processes. Those transition rules are defined in PVS as follows:

```
synpara(X,Q,R):
    EXISTS w1,w2: syn_wtrans(w1)(Q,nul) AND
                  syn_wtrans(w2)(R,nul) AND
                  syn_parallel(w)(w1,w2) AND P1 = nul
 OR EXISTS (a:normal,w1,Q1): X(a) AND ntrans(a)(Q,Q1) AND
                  syn_wtrans(w1)(R,nul) AND
                  w = bottom AND P1 = nul
 OR EXISTS (a:normal,w1,R1) : X(a) AND ntrans(a)(R,R1) AND
                  syn_wtrans(w1)(Q,nul) AND
                  w = bottom AND P1 = nul
 OR EXISTS (a1,a2:normal,Q1,R1): X(a1) AND X(a2) AND a1 /= a2 AND
                  ntrans(a1)(Q,Q1) AND ntrans(a2)(R,R1) AND
                  w = bottom AND P1 = nul,


csynpara(X,QQ,RR):
    EXISTS Q1,R1,w1,w2 : syn_wtrans(w1)(QQ,Q1) AND
                  syn_wtrans(w2)(RR,R1) AND syn_parallel(w)(w1,w2) AND
                  w /= bottom AND P1 = synpara(X,Q1,R1)
 OR EXISTS (a:normal,w1,QQ1,R1): X(a) AND
                  ntrans(a)(QQ,QQ1) AND syn_wtrans(w1)(RR,R1) AND
                  w = bottom AND P1= nul
 OR EXISTS (a:normal,w1,Q1,RR1): X(a) AND
                  syn_wtrans(w1)(QQ,Q1) AND ntrans(a)(RR,RR1) AND
                  w = bottom and P1 = nul
 OR EXISTS (a1,a2:normal,QQ1,RR1):
                  X(a1) AND X(a2) AND a1 /= a2 AND
                  ntrans(a1)(QQ,QQ1) AND ntrans(a2)(RR,RR1) AND
                  w = bottom AND P1 = nul
```

With the introduction of the ⊥, we can extend the existing definitions of terminal transi-

tions. The transition rules defined earlier assumed $\omega \neq \bot$. We can extend each transition rule adding the term: $\omega \neq \bot \Rightarrow \dots$. For example, the terminal transitions of standard sequential composition can extended as follows:

```
seq(Q,R) : w /= bottom IMPLIES
  (wtrans(tick)(Q,P1) AND  wtrans(w)(R,nul)
   OR
   w /= tick AND wtrans(w)(Q,nul)),
```

6.7.3   Proving Synchronous Semantic Relationships

With the extended definitions of the semantic models, we can now define the lemmas showing the correspondence between the semantic models.

Standard Processes

The lemma for standard processes (Lemma 5.7, page 75) is defined as follows:

```
synpara_lemma : LEMMA
    trans_trace((s,w))(synpara(X,P,Q),nul) =
      EXISTS (s1,w1,s2,w2) :
        full_parallel(X)((s1,w1))((s2,w2))((s,w)) AND
        trans_trace((s1,w1))(P,nul)              AND
        trans_trace((s2,w2))(Q,nul)
```

It is mentioned in an earlier chapter that during synchronisation processes might deadlock and terminate with a bottom event representing the partial behaviour from the composition. We also consider it in the proof of the above lemma.

Compensable Processes

We define two separate lemmas for the parallel operator for the compensable processes. In one lemma, we consider those cases where processes will synchronise or interleave but do not introduce a $\bot$. In the other lemma, we consider those cases where processes fail to synchronise and we get a partial behaviour from the composition.

First, we define the lemma considering that processes will not fail to synchronise and hence, there is no bottom event in the derived trace:

$$(PP \parallel_X QQ) \xrightarrow{t} R = \exists\, p, q, P, Q \cdot t \in (p \parallel_X q) \,\wedge\, last(t) \neq \perp$$
$$\wedge \; PP \xrightarrow{p} P \,\wedge\, QQ \xrightarrow{q} Q \,\wedge\, R = (P \parallel_X Q)$$

The corresponding definition in PVS is given as follows:

```
    csynpara_lemma : LEMMA
        ftrans_trace((s,w))(csynpara(X,PP,QQ),R) =
            EXISTS (s1,w1,s2,w2,P,Q):
                w /= bottom AND
                full_parallel(X)((s1,w1))((s2,w2))((s,w)) AND
                ftrans_trace((s1,w1))(PP,P)              AND
                ftrans_trace((s2,w2))(QQ,Q)              AND
                R = synpara(X,P,Q)
```

Next, we define the lemma where compensable parallel processes fail to synchronise during their synchronisation. We get partial behaviour from the composition and the lemma for this situation is defined as follows:

$$(PP \parallel_X QQ) \xrightarrow{t} 0 = \exists\, p, q, P, Q \cdot t \in (p \parallel_X q) \,\wedge\, last(t) = \perp$$
$$\wedge \; p \in T(PP) \,\wedge\, q \in T(QQ)$$

We define the lemma in PVS in the same way as the previous lemma. The main difference is that the derived trace now ends with a bottom representing the partial behaviour and compensations are not accumulated after termination. The lemma is defined in PVS as follows:

```
    lema_bot : LEMMA
        ftrans_trace((s,w))(csynpara(X,PP,QQ),nul) =
         EXISTS (s1,w1,s2,w2,P,Q):
            w = bottom AND
            full_parallel(X)((s1,w1))((s2,w2))((s,w)) AND
            ftrans_trace((s1,w1))(PP,P)              AND
            ftrans_trace((s2,w2))(QQ,Q)
```

6.8 Discussion

Our main purpose of this experiment of mechanisation was to find a feasible mechanisation, where we can mechanise both semantic models, and prove their relationship by following the steps as shown in the hand proofs, so in a later stage, the proof techniques can be applied to proofs without any support from the hand proof. To meet

this goal, after mechanising the semantic models, we have performed our experiments to verify whether we can follow the proof steps as shown in the hand proofs to prove some lemmas. After successfully mechansing the sequential, and the parallel composition of standard processes, our next challenge was to mechanise the proofs for the compensation pair, and the transaction block, because their proofs require the use of mutually defined datatypes. One of our main challenges was to mechanise the synchronous composition of processes, because no hand proof has been defined for them, and each proof consists of several cases, which make the proofs difficult to handle.

Although defining process algebras in PVS is not new, our novelty in this experiment is that, we have defined the cCSP process algebra, and the two semantic models, and we have proved a relationship between these semantic models. We have followed similar steps to mechanise the proofs as shown in the hand proofs, and it has given us the confidence in our language definitions. In this section, we discuss our experiences in PVS. We discuss the proof techniques that we have followed in this chapter. We also discuss the limitations of the current techniques, and the difficulties while doing the proofs. Finally, we mention the lessons that we have learned from the experiment.

We have defined the operators in terms of their semantic models, and laws of these operators can be proved from these semantic definitions. We have defined the lemmas by using a boolean equality between the derived trace, and the derived equation from the definition of the original trace. During the proofs, for both the base, and the inductive cases, this equality has been translated into double implication. For example, a lemma defined of the form, $A = B$ will have the form $A \Leftrightarrow B$. As a result, we have to prove $A \Rightarrow B$ as well as $B \Rightarrow A$.

For example, consider Lemma 4.6 for the sequential composition. First, we apply induction over s, and we get the base, and the inductive case. Only the base case is shown here:

```
seq_lemma :
  |-------
{1}   FORALL (P, Q: stand, s: list[normal], w: terminal):
        trans_trace(s, w)(seq(P, Q), nul) =
          (EXISTS (s1, w1, s2, w2):
              (s, w) = seq((s1, w1), (s2, w2)) AND
              trans_trace(s1, w1)(P, nul) AND trans_trace(s2, w2)(Q, nul))
Rule? (induct "s")
Inducting on s on formula 1,
this yields  2 subgoals:
seq_lemma.1 :
  |-------
{1}   FORALL (P, Q: stand, w: terminal):
```

```
        trans_trace(null, w)(seq(P, Q), nul) =
          (EXISTS (s1, w1, s2, w2):
              (null, w) = seq((s1, w1), (s2, w2)) AND
               trans_trace(s1, w1)(P, nul) AND trans_trace(s2, w2)(Q, nul))
```

In order to carry on the proof, the equations are then skolemized, and flattened, and finally, `iff`, which converts the boolean equality into implication.

```
Rule? (iff)
Converting top level boolean equality into IFF form,
Converting equality to IFF,
this simplifies to:
seq_lemma.1 :
  |-------
{1}   trans_trace(null, w!1)(seq(P!1, Q!1), nul) IFF
        (EXISTS (s1, w1, s2, w2):
            (null, w!1) = seq((s1, w1), (s2, w2)) AND
             trans_trace(s1, w1)(P!1, nul) AND trans_trace(s2, w2)(Q!1, nul))
```

We then get two implications to be proved after splitting the subgoal.

```
Rule? (split)
Splitting conjunctions,
this yields  2 subgoals:
seq_lemma.1.1 :
  |-------
{1}   trans_trace(null, w!1)(seq(P!1, Q!1), nul) IMPLIES
        (EXISTS (s1, w1, s2, w2):
            (null, w!1) = seq((s1, w1), (s2, w2)) AND
             trans_trace(s1, w1)(P!1, nul) AND trans_trace(s2, w2)(Q!1, nul))
seq_lemma.1.2 :
  |-------
{1}   (EXISTS (s1, w1, s2, w2):
           (null, w!1) = seq((s1, w1), (s2, w2)) AND
            trans_trace(s1, w1)(P!1, nul) AND trans_trace(s2, w2)(Q!1, nul))
        IMPLIES trans_trace(null, w!1)(seq(P!1, Q!1), nul)
```

For the former case, we apply the same proof steps that we have already applied in hand proofs. However, for the later case, the same proof steps cannot be applied directly. After analysing the proof steps of the first case, and by using the trace, and the operational rules, it is possible to prove the second case.

The mechanisation of the proofs has helped us identifying some definitions that were not explicitly defined in the hand proofs. Some supporting lemmas have also been discovered in the inductive proofs that have helped to carry on the corresponding proofs. Some of them are specific to particular lemmas, whereas two lemmas are applied in all the proofs. The lemma `nul_eq` states that, a transition by a trace, where the trace has only terminal event, is same as a terminal transition. The PVS 'grind' command can prove this lemma. Another lemma is 'th1', which states that, all the process terms have a trace. From the original definition of traces, we know that a trace of a process has at least a terminal event, and it can be written as,

$$
\begin{aligned}
\langle \omega \rangle \quad &\in \quad T(P) \\
\Rightarrow \langle \omega \rangle \quad &\in \quad DT(P) \\
&= \quad P \xrightarrow{\langle \omega \rangle} 0
\end{aligned}
$$

The lemma can be proved by following the above derivation. In the inductive case of each lemma, we have used a rule, which has derived from the transition rules of the corresponding operator. The general form of the rule can be defined as follows:

$$
P \xrightarrow{\langle a \rangle t} 0 \quad = \quad \exists\, P' \cdot P \xrightarrow{a} P' \wedge P' \xrightarrow{t} 0
$$

We have discussed that, in mechanisation, we have to prove some steps, which were not required in the hand proofs ($B \Rightarrow A$), and to support this step, we have defined some supporting lemmas, and axioms. For example, consider Lemma 4.6 for sequential composition of standard processes, a lemma has been defined from the following trace rules:

$$
\begin{aligned}
(\langle \omega \rangle = p \,;\, q) \quad &= \quad (p = \langle \checkmark \rangle \wedge q = \langle \omega \rangle \\
&\vee \quad p = \langle \omega \rangle \wedge \omega \neq \checkmark)
\end{aligned}
$$

The corresponding PVS definition is given as follows:

```
empty_trace : LEMMA
    (null,w)= seq((s1,w1),(s2,w2)) =
          (null?(s1) AND w1 = tick AND null?(s2)  AND w = w2)
      OR  null?(s1) AND w1/= tick AND w = w1
```

For the inductive step, we have defined another lemma from the following trace rule:

$$
\begin{aligned}
(\langle a \rangle t = p \,;\, q) \quad &= \quad (\exists\, p' \cdot p = \langle a \rangle p' \wedge t = p' \,;\, q \\
&\vee \quad p = \langle \checkmark \rangle \wedge q = \langle a \rangle t)
\end{aligned}
$$

For the convenience of proof, we have defined the lemma in PVS as follows:

```
eq3 : LEMMA
   (cons(a,s),w) = seq((s1,w1),(s2,w2)) =
        (NOT null?(s1) AND cons?(s1) AND
        car(s1) = a AND (s,w) = seq((cdr(s1),w1),(s2,w2)))
     OR (null?(s1) AND w1 = tick AND NOT null?(s2) AND cons?(s2) AND
        car(s2) = a AND (s,w) = seq((s1,w1),(cdr(s2),w2)))
```

The proofs for the compensation pair, and the transaction block were crucial in our experiments. Due to their behaviour, it was harder to prove them than the other lemmas. Like other lemmas, we have defined some axioms for them to support their proofs. For the compensation pair, we have used a rule derived from the trace semantics, defined as follows:

$$((\langle a \rangle t, t') \;=\; (p \div q)) \quad = \quad \exists\, p' \cdot p = \langle a \rangle p' \,\wedge\, (t, t') = (p' \div q)$$

For the transaction block, we have used two rules derived from the trace semantics, as follows:

- $(\langle a \rangle t \;=\; [\,p, p'\,]) \quad = \quad \exists\, p'' \cdot p = \langle a \rangle p'' \,\wedge\, t = [\,p'', p'\,]$
  $$\vee \quad p = \langle\,!\,\rangle \,\wedge\, \exists\, p'' \cdot p' = \langle a \rangle p'' \,\wedge\, t = [\,p, p''\,]$$

- $(\langle \omega \rangle \;=\; [\,\omega 1, \omega 2\,]) \quad = \quad \omega 1 =\,! \,\wedge\, \omega = \omega 2$
  $$\vee \quad \omega 1 = \checkmark \,\wedge\, \omega = \omega 1$$

6.8.1 Limitations

Although we have successfully mechanised most of the operators of cCSP, and proved several lemmas showing the relationship between the two semantic models, our mechanisation techniques have several limitations. There are also several difficulties that we have faced during mechanisation. In this section, we describe the limitations, and difficulties in our mechanisation techniques, and we outline their impacts in the proofs.

In the typechecking of the operational rules, two TCCs are not discharged automatically, which are generated to verify the subterm relation of the block operator. Although by adding Q1<<P, and Q<<P in `wtrans`, and `ntrans` respectively, can remove those TCCs, it will introduce inconsistencies in the definitions as well as in the proofs, and it will not allow the definition of block inside a block. We were unable to find a proper solution of this problem. As we know that the current definition allows block inside block, we have proved the lemma leaving those TCCs unfinished.

We have defined the terminal transitions of all the process terms in a single definition, `wtrans`. We have first defined the terminal transitions, and then the normal transitions, where the terminal transitions have been used in the definitions of normal transitions.

Unlike the terminal transitions of other process terms, the terminal transitions of the synchronised processes have both normal, and terminal transitions. Hence, it is not possible to included these transition rules in the existing definitions, and a separate definition has been given for them. This definition does not have any observable effect in the current proofs, where our main concern is to prove the lemma showing a relationship between the two semantic models. But, in this definition, the synchronised terminal transition is considered only at the outermost level of a composition, which is not suitable for a general case. An extensive investigation is required to find a solution, which can be applied to the general case.

In the proof of the lemma '`csynpara_lemma`', only those transitions are considered, where the synchronisation does not introduce a bottom. The terminal transitions for the synchronised compensable processes have four possible transitions, and three of them introduce bottom. When this definition is expanded in the proof of the lemma, those three transitions contradict the definition of the lemma, and it is not possible to continue the proofs for those cases. As those three cases are not considered in the proofs, a possible solution of the problem that we have taken is by removing those transitions while proving this particular lemma.

We have mentioned that, for each lemma of the form $A = B$, in PVS, we have to prove $A \Rightarrow B$, and $B \Rightarrow A$, where the proofs of the former case follow the steps in the hand proofs, and for the later case, the proof steps follow the opposite direction of the former case. If we check any proof tree shown in the Appendix C, we can see that each tree has two subtrees: one for the base case, and another for the inductive case. For each case, there are also two subtrees. The left part represents the proof steps in the hand proofs, and the right part represents the proof steps that have not done in the hand proofs ($B \Rightarrow A$). For all the lemmas, especially, those for synchronous composition, the mechanical proofs require several times more steps than that of hand proofs. The mechanical proofs overcome several limitations in the hand proofs, and able to handle complex cases, like, synchronisation, and it might be a trade-off between these advantages, and the number of proof steps.

Although we have successfully mechanised both asynchronous, and synchronous composition of processes, we are at an early stage in mechanising the synchronous composition of compensable processes. With some limitations, we have mechanised the proofs of the lemmas for the synchronous composition of compensable process, but to make the proofs applicable for a general case, it is required to pursue further research in this direction.

6.8.2   Lessons Learned

We started mechanising the semantic models and their relationship in order to investigate the feasibility of the mechanisation process. Feasible mechanisation will allow

us to perform the proofs for the future extensions of the language without performing any proof by hand. Considering this, while deriving the proofs for the synchronous composition of processes, we have avoided describing the proof steps in the hand proofs (Chapter 5). In this chapter, the proofs for the sequential and the asynchronous compositions have been mechanised by following the steps as in the hand proofs. The proof steps of the lemmas are similar. After being familiar with the proof steps of the asynchronous composition, the proofs for the synchronous composition have been carried out in PVS, without any support from the proofs by hand.

It is not necessary to master all the proof commands in order to use the PVS prover for a particular purpose. All of our lemmas have similar proof steps, and same proof commands have been used in many proofs. Some proof commands are the induction scheme, the case analysis, the quantifier instantiation, the skolemization, the propositional simplification rules (`flatten, split`). The PVS prelude theories provide a good source of background mathematics, and a rich source of examples.

In an earlier stage, we had defined separate transition rules for standard, and compensable processes, which did not allow us to prove some lemmas. Adding the null process in the definition of the process terms, allowed us to combine the transition rules for both standard and compensable processes, and prove the lemmas. The user defined recursive datatype definition of PVS is a very useful mechanism to define mutually recursive datatype, which permitted us to define the cCSP process terms in PVS.

In the hand proofs, it is easy to be imprecise about recursion, and typing of the rules. The mechanisation forces to be strict about datatypes, and recursion. This helped us to define the theorems, and the lemmas in a systematic way, and to prove all the lemmas by following a similar fashion. The mechanisation also helped us identifying some lemmas which were not explored in the hand proofs, e.g., `nul_eq, c_nul_eq, add2trace` etc. The mechanisation of the semantic models and their relationships also deepen our understanding of the semantic models for both standard and compensable processes.

6.9   Related Work

A lot of existing work has been devoted to mechanisation by using the generic theorem prover PVS. However, most of them are aimed at concrete applications and very few of those had their focus in theoretical issues, especially on mechanising process algebras or their semantics. This is the first attempt at mechanising the cCSP process algebra by using PVS and to the best of our knowledge this is also the first attempt at mechanising both operational and denotational models of an algebra as well as their relationships by using PVS.

One of the contributions most related to our work is by Basten and Hooman in [12],

where the focus is on the use of a general purpose proof checker, e.g., tool support for the proof of theoretical properties of an ACP-style process algebra [10] . The idea is to apply equational reasoning. Mechanical support for both verification of concrete applications and proving theoretical properties of the process algebra are investigated. The significant similarity with our work is that the process terms are defined by means of abstract datatypes of PVS which helps to use induction on the structure of the terms. After defining the process terms, process equivalence has been drawn along with proving some algebraic properties. A comparison had also been made between defining process terms as an uninterpreted type with defining them using an abstract datatype. The main difference with our work is that they used algebraic semantics to define the processes, and process equivalence is drawn by axiomatic definition, whereas we use a different process algebra and define both operational and denotation semantics and process equivalence is defined by using trace equivalence.

It is already mentioned that PVS has been used in [38, 39, 40] to mechanise the trace semantics of CSP. Their goal is to verify an authentication protocol specified in CSP to overcome errors in the manual verification as well as improve the scalability of the approach. The mechanisation is based on a semantic embedding of CSP. The traces are defined by using a list of events and processes are defined by prefix-closed sets of traces. The important distinction with the present work is that cCSP traces are non-empty and completed and processes are defined accordingly. So the trace definition mentioned above is not applicable in our definitions. Also our research focus is in proving the theoretical properties, especially proving the relation between semantic models, not in concrete applications.

Camilleri [30] showed how to mechanise a subset of the CSP operators by using the theorem prover HOL [52]. The trace model for a subset of the CSP operators was mechanised in HOL. Initially, events, alphabets and traces are defined and then CSP operators are defined in terms of their trace semantic models. And later laws related to the operators are proved from the sematic definition. In contrast to our approach no syntax is defined at this stage and operators are defined directly in HOL. Syntax is defined later and the semantics of the language is shown based on the already defined semantics. A similar work for the π-calculus can be found in [71].

One of our main goals is to explore the ways of incorporating process algebra in a general purpose theorem prover. In that respect, a closely related research on the tool support for a process algebra shown in [55], where a CSP-like algebra, called DI-Algebra [61] is formalised in HOL. The algebra is used to reason about synchronous circuits. Process syntax and algebraic laws are defined, but no semantics are defined. Tactics are defined for both syntactic and algebraic approach. In the syntactic approach operators in the algebra are defined inductively whereas in the algebraic approach processes are defined as functions and axioms are used to define their properties.

Chapter 7

Case Study

The cCSP language has been developed with the intention of modelling long running business transactions. We have developed the semantic models along with establishing their relationship in previous chapters. In this chapter, we present case studies by modelling business transactions using cCSP constructs. Modelling these business processes shows the expressiveness of cCSP constructs and shows how the compensations are orchestrated and at the same time allows us to identify the possible areas to improve the language.

7.1 Introduction

While defining business processes in cCSP, a process is described in terms of its interactions with its environment or other processes. The interactions are described by using atomic actions via channels as in standard CSP [59]. In order to model business processes using the cCSP constructs, we add some constructs to the language which are considered as syntactic sugar to the language.

First, we define communication between processes. As defined in standard CSP, a communication is an event described by a pair $c.v$ where c is the name of the channel on which communication takes place and v is the value of the message which passes.

A construct can be defined to allow an input on channel in of any item x in a set M and the value x determines the subsequent behaviour:

$$in?x : M \; ; Q(x) \quad \widehat{=} \quad \square_{x \in M} \; in.x \; ; Q(x)$$

The complement to the input construct is the output which has the form defined as

follows:

$$out!x \quad = \quad out.x$$

All communications, both input and output, take place via channels. When drawing diagrams of processes, the channels are drawn by using arrows in the appropriate direction to define them as input or output and labelled with names of the channels (Figure 7.1).
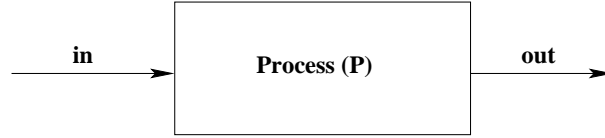

FIGURE 7.1: Channels of a process

Suppose P and Q are processes and c is an output channel of P and an input channel of Q. When these processes are composed to execute concurrently $(P \parallel Q)$, a communication $c.v$ can occur only when both processes engage simultaneously in that event, i.e., whenever P outputs a value v on the channel c, Q simultaneously inputs the same value.

The cCSP choice operator is a binary operator. While modelling the business transactions we use the indexed version of the operator, which is defined as: $\Box_{x \in S} P_x$, e.g.,

$$\Box_{x \in \{S_1, S_2 \ldots S_n\}} P_x \quad = \quad P_{S_1} \Box P_{S_2} \ldots \Box P_{S_n}$$

In the composition $P \parallel Q$, processes P and Q interleave with each other and when three processes are placed in parallel, it does not matter in which order they are put together.

$$P \parallel (Q \parallel R) \quad = \quad (P \parallel Q) \parallel R$$

In the composition $P \parallel_X Q$, processes P and Q synchronise on events from the set X.

In the compensation pair, we also use I/O parameters. By following the definition of standard CSP, a parameterised compensation pair can be defined as follows:

$$(A?x \div B.x) \; ; P(x) \quad = \quad \Box_{x \in S} (A.x \div B.x) \; ; P(x)$$

Here S is finite.

When we define a transaction block, the process inside the block is a compensable process. In the examples presented in this chapter, some processes inside a transaction block are explicitly not defined as compensable processes where the definition is implicit to the process definition (e.g., paired with a *Skip*).

7.1.1 Chapter Structure

In the rest of the chapter, we first model an example of a car broker web service that provides online support for customers to buy cars and to arrange loans for these. The car broker web service uses two other web services: one for finding a better quote for the ordered car and another for arranging loans. We specify their behaviour after describing the behaviour of the car broker web service. We provide two versions of the car broker web service. These examples differ from our earlier example using cCSP (Section 2.6) because of the definition of synchronisation. Finally, we discuss our experiences while modelling these case studies and conclude the chapter.

7.2 Car Broker Web service

A car broker web service negotiates car purchases for buyers and arranges loans for these. The car broker uses two separate web services: a **Supplier** to find a suitable quote for the requested car model and a **Lender** to arrange loans. Each web service can operate separately and can be used in other web services. In the following sections, first we describe the car broker web service and then describe the two other web services. We show how these web services can be composed together for the car broker web service. The original car broker example can be found in [115, 116]. In this case study, we mainly consider how processes are communicating with each other and how compensation is handled when an interruption is occurred. We abstract several details from our description, e.g., how a supplier finds suitable quote for a car model, how a broker selects a quote from several available quotes, how a lender decides to select a loan request, the details a buyer request etc.

7.2.1 A Car Broker Web Service

We model a car broker web service **Broker**. It provides online support to customers to negotiate car purchases and arranges loans for these. A buyer provides a need for a car model. The broker first uses its business partner **Supplier** to find the best possible quote for the requested model and then uses another business partner **LoanStar** to arrange a loan for the buyer for the selected quote. The buyer is also notified about the quote and the necessary arrangements for the loan. Both **LoanStar** and the **Buyer** can cause an interrupt to be invoked. A loan can be refused due to a failure in the loan assessment and a customer can reject the loan and quoted offer. In both cases, there is a need to run the compensation, where the car might have already been ordered, or the loan has already been offered. The behaviour of the **Broker** web service is depicted in Figure 7.2.
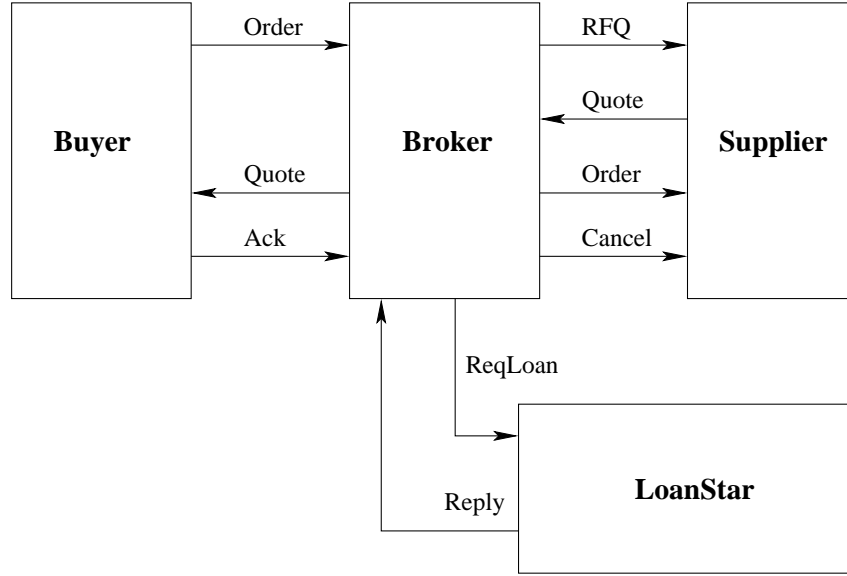
FIGURE 7.2: Architectural view of Car Broker web Services

A separate description of the other two web services will be given later in this chapter. We also add the abstract behaviour of the buyer in order to show the interaction with the **Broker** web service.

The first step of the transaction is a compensation pair, where the primary action is to receive an order from the buyer and the compensation is to cancel the order. M is used to represent the finite set of car models ranged over by m. After receiving the order, it is then passed to the process **ProcessOrder** to perform the rest of the transaction.

$$
\begin{aligned}
\textbf{Broker} \quad &\widehat{=} \quad (\mathit{Order?m : M \div CancelOrder.m}) \,; \textbf{ProcessOrder(m)} \\
\textbf{ProcessOrder(m)} \quad &\widehat{=} \quad \mathit{RFQ.m} \,; \mathit{Quote?q : \mathbb{F}\ Q} \,; \\
& \qquad \square_{c \in q} \bullet \Big((\textbf{Sendorder(c)} \parallel \textbf{Loan(a)}) \parallel \textbf{SendQuote(c)} \Big) \\
\textbf{SendOrder(c)} \quad &\widehat{=} \quad (\mathit{Order.c \div Cancel.c}) \\
\textbf{Loan(a)} \quad &\widehat{=} \quad (\mathit{ReqLoan.a : Amt \div CancelLoan.a}) \,; \\
& \qquad (\mathit{Reply?Accept} \,; \mathit{SKIPP} \\
& \qquad \quad \square\ \mathit{Reply?Reject} \,; \mathit{THROWW}) \\
\textbf{SendQuote(c)} \quad &\widehat{=} \quad \mathit{Quote.c} \,; \ (\mathit{Ack?Accept} \,; \mathit{SKIPP} \\
& \qquad \qquad \qquad \square\ \mathit{Ack?Reject} \,; \mathit{THROWW})
\end{aligned}
$$

After receiving an order for a car from the **Buyer**, the **Broker** first requests the **Supplier** for available quotes (RFQ) and then selects a quote from the received quotes (Quote). We abstract away from the details of how decisions are made. The **Broker** then arranges a loan for the quoted car by requesting a loan from **LoanStar**. The amount of loan to be requested is decided from the selected quote and then passed to the process **Loan**. It requests loan from **LoanStar** and it is either accepted or rejected.

In reality, orders have unique reference and loans, etc are linked to that; here, we are mainly considering how processes are communicating with each other abstracting other details from the description. In the case where the loan is accepted, it is assumed that the loan provider starts its processing to arrange the loan. If the loan cannot be provided then an interrupt is thrown to cancel the actions that already took place. A compensation is added to *ReqLoan* (*CancelLoan*) so that in the case of a failure in a later stage the compensation can be invoked to cancel the event.

The buyer is also notified of the quote for the selected car (**SendQuote**). The **Broker** receives an acknowledgment (*Ack*) from the buyer that is either accepting the quote or rejecting the quote. In the case of rejection, an interruption is thrown to cancel the transaction and run the appropriate compensation. The processes **SendQuote**, **Loan** and **SendOrder** do not have any synchronisation between them and they interleave with each other. An interrupt thrown from either the buyer or the lender can occur before or after ordering the car to the supplier. In either case, the compensation mechanism takes care of it and the proper compensations will run.

We define only an abstract behaviour of the buyer where the buyer first sends an order for a car to the broker. Then after receiving the selected quote from the broker, the buyer either accepts or rejects the quote.

$$
\begin{aligned}
\textbf{Buyer} \quad &\widehat{=} \quad Order.m : M \; ; Quote?q : Q \; ; \square_{r \in R} \, Ack.r \\
R \quad &= \quad \{Accept, Reject\}
\end{aligned}
$$

The behaviour of the car broker web service is then defined by combining the behaviour of **Broker, Buyer, Supplier**, and **LoanStar**.

$$
\begin{aligned}
\textbf{System} \quad &\widehat{=} \quad \Big(\textbf{Buyer} \parallel_A (\lfloor \textbf{Broker} \rfloor \parallel_{\textbf{B}} \textbf{Supplier}) \Big) \parallel_{\textbf{C}} \textbf{LoanStar} \\
A \quad &= \quad \{ \, Order, Quote, Ack \, \}, \\
B \quad &= \quad \{ \, RFQ, Quote, Order, Cancel \, \} \\
C \quad &= \quad \{ \, ReqLoan, Reply \, \}
\end{aligned}
$$

We describe the behaviour of the broker and only give abstract behaviour of the other web services. The **Broker** is defined as a compensable process within a transaction block where the compensations attached to the **Broker** are handled inside the block, keeping the details hidden from outside the block. **Supplier** and **LoanStar** are standard processes. Detailed descriptions of both **LoanStar** and **Supplier** are given later in the chapter. Both of these web services are defined as separate web services and in this example, they are composed together.

7.2.2 A separate version Car Broker web service

In this section, we present a slightly different version of the earlier description of the car broker web service. Giving two separate specifications of same web service allows us to experiment with the expressiveness of the cCSP language. In particular, we show how compensations are handled for concurrently executing compensable processes within a transaction block instead of between transaction blocks as shown in the previous version.
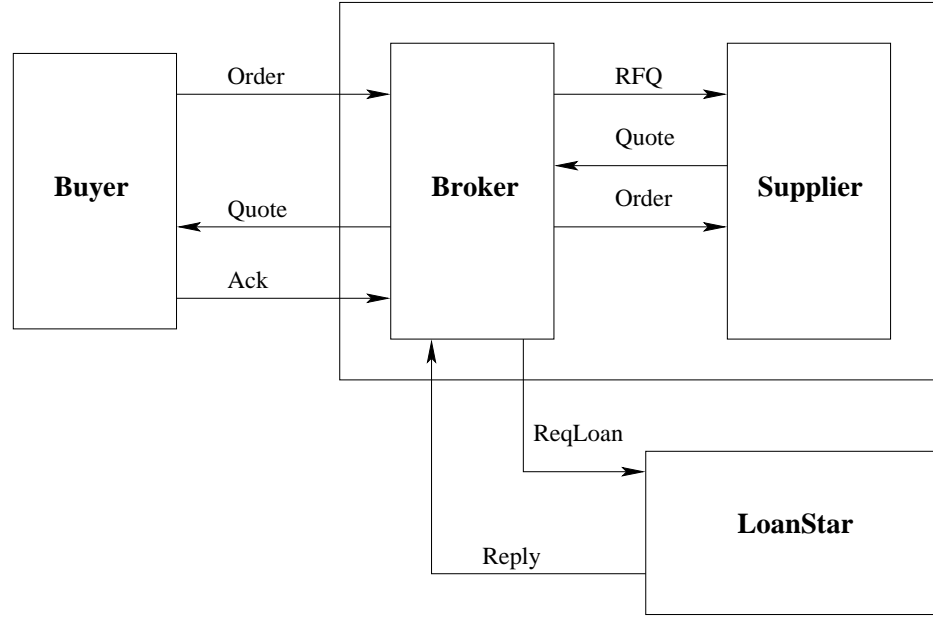

FIGURE 7.3: Another version of CarBroker web services

Figure 7.3 represents the alternative description of the Car Broker web services. The main distinction from previous example is that both **Broker** and **Supplier** execute concurrently within a transaction block. A box is drawn surrounding the **Broker** and the **Supplier** to denote the scope of the transaction block. When an interrupt is thrown in any process within a transaction block, the compensation will run inside the block. In the present example, both **Broker** and **Supplier** are compensable processes and they execute concurrently. When an interrupt is raised in any of the processes, the forward execution of the concurrent processes will terminate and the corresponding compensations will run. Here in the example, an interrupt can be raised either by the **Buyer** by rejecting the quote sent by the **Broker** or by the **LoanStar** by rejecting the loan request for the quoted car. In either case, the **Supplier** will terminate yielding an interrupt thrown by the **Broker** and compensations from both **Broker** and **Supplier** will run in parallel. Unlike in the previous example, there is no need for a separate cancel message from **Broker** to **Supplier** as the interrupt will be yielded automatically by the **Supplier** within the block. After getting a definite order from the **Broker**, the state (e.g., Database) of the **Supplier** is updated to reflect the order and a compensation can be attached to it, which will run whenever an interrupt is raised by any concurrent process inside the transaction block to undo the effect.

As both processes are within a block, we do not need a separate cancel message from the **Broker** to the **Supplier** to cancel the transactions. As a result, we change the **SendOrder** process for the current example, where *cancel* is removed from the compensation.

$$\textbf{SendOrder(c)} \quad \widehat{=} \quad (Order.c \div Skip)$$

By including both **Broker** and **Supplier** in a transaction block, the overall behaviour of the car broker web service is defined as follows:

$$\textbf{System} \quad \widehat{=} \quad (\textbf{Buyer} \parallel_A [\,\textbf{Broker} \parallel_B \textbf{Supplier}\,]) \parallel_C \textbf{LoanStar}$$

$$A \quad = \quad \{\,Order, Quote, Ack\,\}$$
$$B \quad = \quad \{\,RFQ, Quote, Order\,\}$$
$$C \quad = \quad \{\,ReqLoan, Reply\,\}$$

7.2.3 A Lender Web Service

A loan service is a common example of a business process (please refer to [115] for a full description). We assume a lender web service **LoanStar**, that offers loans to online customers. A customer submits a request for an amount to be loaned along with other required information. **LoanStar** first checks the loan amount and if the amount is £10,000 or more, then **LoanStar** asks its business partner **FirstRate** to thoroughly assess the loan. After a detailed assessment of the loan, **FirstRate** can either approve the loan or reject the loan.

A full assessment is costly, so if the loan amount is less than £10,000, the loan is evaluated more simply. **LoanStar** asks its business partner **Assessor** to evaluate the risk for the loan. If the associated risk is low then loan is approved, otherwise **LoanStar** asks **FirstRate** to perform a full assessment.

We are giving a simple specification of the lender and are not considering any attached compensation. Therefore, the processes are defined as standard processes. At the top level, the transaction is defined as a sequence of two processes. First, it receives a loan order from the customer and then processes the loan.

$$\textbf{LoanStar} \quad \widehat{=} \quad LoanOrder?a : Amt \,; \textbf{Process}(a)$$

After the request is received from the customer, the requested amount is passed to the process called **Process** to take the necessary steps before arranging the requested loan. It first checks the loan amount in order to determine the type of evaluation that it needs to perform before accepting the loan. We define a process *ChkAmt* which checks the

loan amount in the order to determine whether the amount is over or below the given limit, which is in this case £10,000. Here, *ChkAmt*, *Blow* and *Over* abstract away the details of how the checking has been done.

$$\textbf{Process}(\textbf{a}) \quad \widehat{=} \quad ChkAmt.a \; ; \; (Below.a \; ; \textbf{Assessor}(\textbf{a})$$
$$\square \; Over.a \; ; \textbf{FirstRate}(\textbf{a}))$$

If the loan amount is less than £10,000, then the process **Assessor** will start. It first checks the risk associated with the loan. If the risk is low the loan is approved. If the risk is high then control is passed to **FirstRate** to perform a full assessment. On the other hand, if the amount is higher than or equal to £10,000, then **FirstRate** will start its assessment immediately. After performing a full assessment and depending on the outcome, **FirstRate** either accepts or rejects the requested loan.

$$\textbf{Assessor}(\textbf{a}) \quad \widehat{=} \quad ChkRisk.a \; ; \; (Low.a \; ; Reply.Accept$$
$$\square \; High.a \; ; \textbf{FirstRate}(\textbf{a}))$$

$$\textbf{FirstRate}(\textbf{a}) \quad \widehat{=} \quad Assess.a \; ; \; (Ok \; ; Reply.Accept$$
$$\square \; NotOk \; ; Reply.Reject)$$

In the example, we abstract the details of the behaviour of **Assessor** and **FirstRate**. Both of them can be modelled as a separate web service or as a part of the lender web services.

7.2.4 Elaborating the Supplier Web Service

Here we present an elaborated description of the **Supplier** web service. An abstract view of it was used in the previous examples.

A car supplier web service provides buyers a good deal on car orders. **Supplier** is a supplier that takes orders for a car from buyers. It then sends requests for quotes to a dealer to get available quotes for the requested car model. The dealer collects quotes from all of its associated partners and then accumulates the received quotes and passes them to the car supplier. The better offer is then selected by the car supplier based on lowest price, or earliest delivery date, if price is equal. This offer is then sent to the buyer and at the same time to the dealer as a definite order for the selected model as it would be expected that the buyer will accept the offer. After receiving the quote from the supplier, the buyer can either accept or reject the quote. In the case of rejection from buyer, a compensation is invoked to cancel the order that is sent to the dealer. Here, we give a simple representation of the order receipt and dealer activities and focus on the behaviour of the car supplier in more detail.
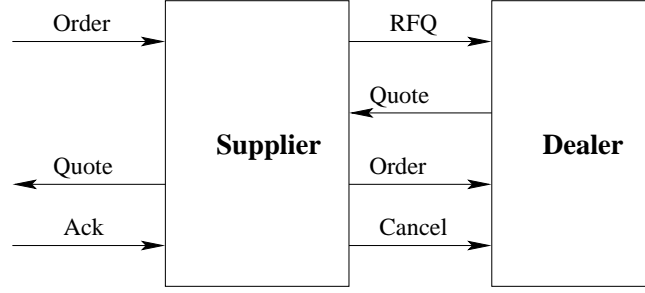
FIGURE 7.4: A car supplier web service

The first step in the transaction is a compensation pair where the primary action is to receive an order from the buyer and then start processing the order. The compensation is to reject the order which will be invoked if there is an interruption at a later stage of the transaction. Here, M is the set of car models.

$$\textbf{Supplier} \quad \widehat{=} \quad \big[\, (Order?m : M \div CancelOrder.m) \,; \textbf{ProcessOrder(m)} \,\big]$$
$$\text{where,} \qquad M = \text{Car Models}$$

Following the compensation pair, the process **ProcessOrder** starts. It first sends a request for a quote (RFQ) to the dealer for available quote for the car model. After receiving a set of quotes from the dealer, one quote is selected by **ProcessOrder**. The selected quote is then sent to buyer (**SendQuote**) and also to the dealer ($Order$) to order the car. A compensation is augmented to the $Order$ which might need at a later stage to compensate the $Order$ activity. Here, Q is used to represent the set of quotes.

$$\textbf{ProcessOrder(m)} \quad \widehat{=} \quad RFQ.m \,; RecQuote?q : \mathbb{P}\, Q \,;$$
$$\square_{c \in q} \bullet \Big((Order.c \div Cancel.c) \parallel \textbf{SendQuote(c)} \Big)$$
$$\text{where,} \qquad Q = \text{Available Quotes}$$

The buyer acknowledges the receipt of a quote by either accepting it or rejecting it. In the case of rejection, an interrupt is thrown, so that the appropriate compensations can be invoked to compensate those activities that did take place. It has been discussed earlier that as **SendOrder** and **SendQuote** interleave with each other, the interrupt from the buyer can be thrown before sending the order to the dealer. The compensations are stored dynamically during the execution of processes, which is in this case empty and compensation mechanism can take care of it. The process **SendQuote** is defined as follows where it first sends a quote and then receives an acknowledge which is either *Accept* or *Reject*:

$$\textbf{SendQuote(c)} \quad \widehat{=} \quad Quote.c \,; \; (Ack?Accept \,; SKIPP$$
$$\square \; Ack?Reject \,; THROWW)$$

The behaviour of **Dealer** and buyer is not fully specified here. Our main focus is on the behaviour of the car supplier.

The behaviour of the supplier system can be defined by composing the behaviour of **Supplier**, **Dealer** and **Buyer**.

$$
\begin{aligned}
\textbf{System} \quad &\widehat{=} \quad \Big(\textbf{Buyer} \parallel_A \textbf{Supplier}\Big) \parallel_B \textbf{Dealer} \\
A \quad &= \quad \{Order, Quote, Ack\} \\
B \quad &= \quad \{RFQ, Quote, Order, Cancel\}
\end{aligned}
$$

7.3 Discussion

In the case studies, we have shown how cCSP constructs can be used to model business transactions. Importantly, we have shown how compensations are orchestrated to model the business processes. The compensations are accumulated during the execution of the processes. The compensations are defined in such a way that when an interrupt occurs at any stage of the transaction, the appropriate compensations (which might be empty when interruption occurs before occurring an event with attached compensation) are executed for the actions that already did take place. For the case studies, we have made the model simpler for ease of understanding.

Previously, we have shown how to model business transactions by using cCSP constructs (Chapter 2). The significant improvement in the current examples is the use of synchronisation to model communication between concurrent processes. We have shown the synchronisation between compensable processes within a transaction block as well as between standard processes and transaction blocks. For example, we modelled two different versions of the **Broker** web service where the distinction is made based on how the synchronisation is defined between the processes. For concurrent compensable processes within a block, we have shown how one process (**Supplier**) catches an interrupt thrown by another concurrent processes (e.g.,**Broker**). The concurrent processes terminate and their appropriate compensations are executed. On the other hand, when both **Broker** and **Supplier** are considered as separate blocks, then an interrupt thrown inside the block will run the attached compensation for that process within the block and it is not observable to other processes outside of the block. Hence, a message (*cancel*) is sent from **Broker** to **Supplier** to terminate the concurrent execution. The compensation attached to the **Supplier** will then run within its own block. Although we only focus on modelling the behaviour of **Broker** in this chapter, it would be a good exercise for our future work to model the behaviour of other processes which we have abstracted in this case study.

This kind of case study gives feedback support about the expressiveness of the language

constructs to model business transactions. Such a case study also provides a better understanding of the language constructs as well as provides direction towards further improvement of the language.

While modelling the business transactions, we have faced several difficulties to represent the real world actions or events in cCSP. We abstracted many real word behaviour from in the case study. We realise that having the facility to model the states of the processes would improve the modelling of the transactions where cCSP can be used to coordinate the interaction of the processes and any state based approach can be used to represent the states. Several research such as [28, 122], have been carried out in this direction and could be a good starting point for our future work.

Chapter 8

Conclusions and Future Work

Process algebras are increasingly being used to provide a formal framework in the modelling of business transactions. In recent years, several process algebras have been proposed in this respect. In this thesis, we emphasise the formal foundation of such a process algebra. Providing semantic definitions for such an algebra is very important for its formal foundation. This thesis has extended the formal semantics of the process algebra, compensating CSP and defined a formal relationship between the semantic models. This concluding chapter summarises the work presented in this thesis and suggests some future research directions.

8.1 Summary

The aim of this thesis has been to strengthen the formal foundations of the cCSP process algebra. Our research contributes to the following improvements of the cCSP semantic models:

- **Extending Semantic Models:** Compensating CSP was defined by giving its trace semantics [27]. Starting from the trace semantics, we extend the semantic model by defining an operational semantics (Chapter 3) for the algebra. The operational semantics is defined by using labelled transition systems which give a closer view of the executions of the states of a program. Transition rules are defined for both standard and compensable processes. We show how compensations are managed for compensable processes. Both sequential and parallel operators are redefined so that attached compensations are accumulated in an appropriate order after termination. We use small-step semantics to define the transition rules. Small-step semantics describes how the individual steps of a computation take place and it can describe the evaluation of both terminating and non-terminating

behaviours. It can be regarded as one-state-at-a-time recipe for computing the transition system of any process.

Initially, the parallel operator for cCSP was defined to interleave over observable events and synchronise over terminal events. We extend the semantic models further to define the transition rules for the parallel operator where processes can synchronise over synchronising observable events (Chapter 5). We introduce the notion of *partial behaviour* (Section 5.2) which is analogous to prefixes in standard CSP, to model the behaviour of synchronised processes that lead to deadlock. How the attached compensations of a compensable processes are affected by the partial behaviour is also described for concurrent processes (Section 5.4.2).

Apart from defining the transition rules, the operational rules were encoded into Prolog to make it executable and animate the specification by using XTL which can later be used to support model checking (Section 3.3).

- **Relating Semantic Models:** Given two different semantic definitions for cCSP, in this thesis, we have defined a relationship between the two semantic models (Section 4.2). The relation shows that traces extracted from the transition rules in the operational semantics correspond to the originally defined traces in the trace semantics. We have defined how to extract traces from the transition rules. Separate definitions are given for standard and compensable processes.

  The correspondence between the derived and original traces is proved by using structural induction. Two levels of induction are applied to prove the correspondence. Supporting lemmas are defined for each operator of the algebra for standard as well as compensable processes. We have shown how the trace operators are applied both at trace and at process level to define these lemmas and help to derive the proofs (Section 4.2.7). The inductive proofs have been carried out completely by hand.

  With the extension of the semantic models to support synchronisation, we have also shown the correspondence between the semantic models for concurrency operator (Section 5.5). Instead of showing the detailed proof steps as for asynchronous processes, we have only outlined how to carry out the proofs. We have defined how to extract partial traces when synchronising processes introduce deadlock, and outlined how to derive their correspondence with the original semantic definition.

- **Mechanising Semantic Models:** The relationship between the semantic models has been mechanised by using the theorem prover PVS (Chapter 6). We used shallow embedding for mechanising the semantic models and their relationships. The process algebra terms are mechanised by using mutually dependant datatypes (Section 6.3.3). The semantic models and the supporting lemmas are defined following a similar approach as in hand proofs. The proofs have been carried out by interacting with the prover. Proof through interaction has been shown to be both fruitful and practical: the language cCSP and the formal environment

provided by PVS enables the construction and reasoning of the semantic models at a relatively high level. Besides, the feedback from the theorem prover is helpful in improving the semantic models.

During mechanising the semantic models and the proofs of their relationship, we follow similar levels of steps as in hand proof. The proofs of all the lemmas follow similar patterns which ease the proof steps. The mechanisation also reveals some of the lemmas that are not explicitly defined in hand proofs.

The mechanisation forces us to be very precise about the typing of rules and applying recursion (mutual recursion). It helps improve structuring the steps in the inductive proofs.

We have experimented with the expressiveness of the cCSP language by modelling web services (Chapter 7). Several web services have been modelled by cCSP constructs and it is shown how these web services can be combined together. Two separate versions of the same web service have been modelled (Section 7.2.1, Section 7.2.2), which allowed to investigate the synchronisation of processes within and between transaction blocks. These case studies also give us the feedback for future improvements of the language.

8.2 Future Work

This thesis is mainly concerned with the development of formal models for an algebra to model business transactions. Whilst some progress has been made in improving the semantic models and proving their properties, there is much scope for further investigation. In this section, we discuss some further research topics arising out from this thesis.

- **Integration with state oriented language:** There has been interest for many years in combining a state based approach (mainly Z [112] and B [3]) with a process algebra (CSP, Timed CSP, CCS), see for instance, [23, 28, 48, 68, 107, 108, 114, 122]. We are particularly interested in the combination of cCSP with the B method. We intend to use cCSP and B in a complementary way. Abstract states and operations of a system can be specified by using the B method. cCSP, on the other hand, can be used to specify the overall coordination of operations. To combine these two approaches, an operation of a B machine can be considered as an event in cCSP terms. Some closely related work is done in [28] to combine CSP and B. Both CSP and B specifications are placed in parallel for their combined meaning and both specifications must synchronise on common events. This work could be the basis of our future work. The combination with a state based approach could help to properly model the states of business transactions (Chapter 7). We

can take advantage of existing tool support, such as the RODIN tool [4], Click'n Prove [5], ProB [65], for the verification and refinement for the B machines.

- **Tool support:** Tool support for animating and model checking of the specifications would significantly improve applicability of the algebra. The CIA [64] (CSP Interpreter and Animator) tool is a Prolog-based implementor for CSP. It was later combined with the ProB [65] (a Prolog-based animator and model checker for B) tool that allows the combined use of CSP and B. We have already encoded the cCSP transition rules in Prolog for animating the specification (Section 3.3). ProB can be easily extended to support cCSP specifications and can serve as a tool for animating as well as model checking the cCSP specification. It can also be extended to support the animation and model checking the specifications written in a combination of cCSP and B.

- **Cancellation semantics for synchronised processes:** In the semantic definition of synchronising processes, we only describe the relative order of forward and compensation behaviour and do not mention their relationship. A theory of cancellation is defined in [27] for compensable processes where the effect of a forward action is cancelled by a compensation action. However, synchronisation was not considered during the definition. It would be interesting to extend the cancellation theory to synchronous processes and investigate the interplay between the forward and compensation behaviour of synchronising processes.

- **Extending language features:** Having a firm grasp of the semantic models, we are now in a better position to extend the language by defining some important operators for a process algebra, such as event hiding, recursion, the distinction between external and internal choice in combination with compensations. In standard CSP the distinction between the two choice operators is achieved by using the Failure/Divergences model which can serve as the basis for our work on cCSP.

Appendix A

Encoding of transition rules in XTL

A.1 Standard Processes

Choice:

```
1. trans(choice(P,Q),A,P1):-
   NOT(terminal(A)),
   trans(P,A,P1).
2. trans(choice(P,Q),A,0):-
   terminal(A),
   trans(P,A,0).
```

Sequential Composition:

```
 1. trans(seq(P,Q),A,seq(P1,Q):-
    NOT(terminal(A)),
    trans(P,A,P1).
 2.trans(seq(P,Q),A,0):-
    terminal(A), NOT(A=tick),
    trans(P,A,0).
 3.trans(seq(P,Q),A,Q1):-
    trans(P,tick,0),
    NOT(terminal(A)),
    trans(Q,A,Q1).
 4.trans(seq(P,Q),A,0):-
    trans(P,tick,0),
```

```
    terminal(A),
    trans(Q,A,0).
```

Parallel Composition:

```
1. trans(par(P,Q),A,par(P1,Q)):-
     NOT(terminal(A)),
     trans(P,A,P1).
2. trans(par(P,Q),A,par(P,Q1)):-
     NOT(terminal(A)),
     trans(Q,A,Q1).
3. trans(par(P,Q),W,0):-
     terminal(W1), trans(P,W1,0),
     terminal(W2), trans(Q,W2,0),
     synch(W1,W2,W).
```

Interrupt Handler:

```
1. trans(inthnd(P,Q),A,inthnd(P1,Q)):-
     NOT(terminal(A)),
     trans(P,A,P1).
2. trans(inthnd(P,Q),A,0):-
     terminal(A), NOT(A=throw),
     trans(P,A,0).
3. trans(inthnd(P,Q),A,Q1):-
     trans(P,throw,0),
     NOT(termial(A)),
     trans(Q,A,Q1).
4. trans(inthnd(P,Q),A,0):-
     trans(P,throw,0),
     terminal(A),
     trans(Q,A,0).
```

Transaction block:

```
1. trans(block(PP),A,block(PP1)):-
     NOT(terminal(A)),
     trans(PP,A,PP1).
2. trans(block(PP),tick,0):-
     trans(PP,tick,P).
```

```
3. trans(block(PP),A,P1):-
     trans(PP,throw,P),
     NOT(terminal(A)),
     trans(P,A,P1).
4. trans(block(PP),A,0):-
     trans(PP,throw,P),
     terminal(A),
     trans(P,A,0).
```

Basic Processes:

```
1. trans(SKIP,A,0):-
     A = tick.
2. trans(THROW,A,0):-
     A = throw.
3. trans(YIELD,A,0):-
     A = yield.
4. trans(YIELD,A,0):-
      A = tick.
```

A.2   Compensable Processes

Compensable Choice:

```
1. trans(cchoice(PP,QQ),A,PP1):-
     NOT(terminal(A)),
     trans(PP,A,PP1).
2. trans(cchoice(PP,QQ),A,P):-
     terminal(A),
     trans(PP,A,P).
```

Compensation Pair:

```
1. trans(cpair(P,Q),A,cpair(P1,Q)):-
     NOT(terminal(A)),
     trans(P,A,P1).
2. trans(cpair(P,Q),tick,Q):-
     trans(P,tick,0).
3. trans(cpair(P,Q),A,SKIP):-
```

```
    termina(A), NOT(A=tick),
    trans(P,A,0).
```

Sequential Composition:

```
1. trans(cseq(PP,QQ),A,cseq(PP1,QQ)):-
    NOT(terminal(A)),
    trans(PP,A,PP1).
2. trans(cseq(PP,QQ),A,P):-
    terminal(A),NOT(A=tick),
    trans(PP,tick,P).
3. trans(cseq(PP,QQ),A,seq(P,Q)):-
    trans(PP,tick,P),
    terminal(A),
    trans(QQ,A,Q).
4. trans(cseq(PP,QQ),A,axseq(QQ1,P)):-
    trans(PP,tick,P),
    NOT(terminal(A)),
    trans(QQ,A,QQ').
5. trans(axseq(PP,Q),A,axseq(PP1,Q)):-
    NOT(terminal(A)),
    trans(PP,A,PP1).
6. trans(axseq(PP,Q),A,seq(P,Q)):-
    terminal(A),
    trans(PP,A,P).
```

Parallel Composition:

```
1. trans(cpar(PP,QQ),A,cpar(PP1,QQ)):-
    NOT(terminal(A)),
    trans(PP,A,PP1).
2. trans(cpar(PP,QQ),A,cpar(PP,QQ1)):-
    NOT(terminal(A)),
    trans(QQ,A,QQ1).
3. trans(cpar(PP,QQ),W,par(P,Q)):-
    terminal(W1), trans(PP,W1,P),
    terminal(W2), trans(QQ,W2,Q),
    synch(W1,W2,W).
```

Appendix B

Correspondence Proof

Here we show the proof of the lemmas which are used in the proof of the theorem presented in the main text in Chapter 4.

B.1 Proof of Lemma 4.7

$$(PP \mathbin; QQ) \xrightarrow{t} R \quad = \quad \exists P, Q, p, q \cdot t = (p \mathbin; q) \wedge PP \xrightarrow{p} P \wedge QQ \xrightarrow{q} Q$$
$$\wedge R = COND\,(last(p) = \checkmark, (Q \mathbin; P), P)$$

Proof :

Basic step: $t = \langle \omega \rangle$

$$PP \mathbin; QQ \xrightarrow{\langle \omega \rangle} R$$
$$= PP \mathbin; QQ \xrightarrow{\omega} R$$
$$= \text{``From derived equation 4.10''}$$

$$\exists P, Q \cdot PP \xrightarrow{\checkmark} P \wedge QQ \xrightarrow{\omega} Q \wedge R = Q \mathbin; P \tag{B.1}$$

$$\vee \ \exists P \cdot PP \xrightarrow{\omega} P \wedge \omega \neq \checkmark \wedge R = P \tag{B.2}$$

From (B.1)

$$\exists P, Q \cdot PP \xrightarrow{\checkmark} P \wedge QQ \xrightarrow{\omega} Q \wedge R = Q \mathbin; P$$
$$= \exists P, Q, p, q \cdot p = \langle \checkmark \rangle \wedge q = \langle \omega \rangle \wedge PP \xrightarrow{p} P \wedge QQ \xrightarrow{q} Q \wedge R = Q \mathbin; P$$
$$= \exists P, Q, p, q \cdot \langle \omega \rangle = (p \mathbin; q) \wedge PP \xrightarrow{p} P \wedge QQ \xrightarrow{q} Q \wedge R = Q \mathbin; P$$

From (B.2)

$$\exists\, P \cdot PP \xrightarrow{\omega} P \;\wedge\; \omega \neq \checkmark \;\wedge\; R = P$$

$$= \;\exists\, P, p \cdot p = \langle\omega\rangle \;\wedge\; \omega \neq \checkmark \;\wedge\; PP \xrightarrow{p} P \;\wedge\; R = P$$

$$= \;\exists\, P, p \cdot \langle\omega\rangle = p \;\wedge\; PP \xrightarrow{p} P \;\wedge\; R = P$$

Therefore, for $t = \langle\omega\rangle$, from (B.1) $\vee$ (B.2)

$$\exists\, P, Q, p, q \cdot \langle\omega\rangle = (p \;;\; q) \;\wedge\; PP \xrightarrow{p} P \;\wedge\; QQ \xrightarrow{q} Q \;\wedge\; R = Q \;;\; P$$

$$\vee \;\exists\, P, p \cdot \langle\omega\rangle = p \;\wedge\; PP \xrightarrow{p} P \;\wedge\; R = P$$

The main difference between the two formula of the above disjunction is whether or not, $last(p) = \checkmark$. The sequence operator handles this in such a way that when $last(p) = \checkmark$, the behaviour of QQ is augmented with behaviour of PP, otherwise the behaviour of QQ is discarded. The above two equations can be combined by the expression $COND$.

$$= \;\exists\, P, Q, p, q \cdot \langle\omega\rangle = (p \;;\; q) \;\wedge\; PP \xrightarrow{p} P \;\wedge\; QQ \xrightarrow{q} Q$$
$$\wedge\; R = COND(last(p) = \checkmark, (Q \;;\; P), P)$$

Inductive step: $t = \langle a\rangle t$

$$PP \;;\; QQ \xrightarrow{\langle a\rangle t} R$$

$$= \;\exists\, RR \cdot (PP \;;\; QQ) \xrightarrow{a} RR \;\wedge\; RR \xrightarrow{t} R$$

$$= \;\text{"From derived equation 4.11"}$$

$$\exists\, PP' \cdot PP \xrightarrow{a} PP' \;\wedge\; (PP' \;;\; QQ) \xrightarrow{t} R \tag{B.3}$$

$$\vee \;\exists\, P, QQ' \cdot PP \xrightarrow{\checkmark} P \;\wedge\; QQ \xrightarrow{a} QQ' \;\wedge\; \langle QQ', P\rangle \xrightarrow{t} R \tag{B.4}$$

From (B.3)

$$\exists\, PP' \cdot PP \xrightarrow{a} PP' \;\wedge\; (PP' \;;\; QQ) \xrightarrow{t} R$$

$$= \;\text{"Inductive hypothesis"}$$

$$\exists\, PP' \cdot PP \xrightarrow{a} PP' \;\wedge\; \exists\, P, Q, p', q \cdot t = (p' \;;\; q) \;\wedge\; PP' \xrightarrow{p'} P \;\wedge\; QQ \xrightarrow{q} Q$$
$$\wedge\; R = COND(last(p) = \checkmark, (Q \;;\; P), P)$$

Here $COND$ expression handles both the cases, whether or not $last(p) = \checkmark$.

$=$ "Removing PP' "

$$\exists\, P, Q, p', q \cdot t = (p' \,;\, q)\ \wedge\ PP \xrightarrow{\langle a \rangle p'} P\ \wedge\ QQ \xrightarrow{q} Q$$
$$\wedge\ R = COND(last(p) = \checkmark, (Q \,;\, P), P)$$

$=\ \exists\, P, Q, p, q \cdot p = \langle a \rangle p'\ \wedge\ t = (p' \,;\, q)\ \wedge\ PP \xrightarrow{p} P\ \wedge\ QQ \xrightarrow{q} Q$
$$\wedge\ R = COND(last(p) = \checkmark, (Q \,;\, P), P)$$

$=$ "By trace rule: $\langle a \rangle t = \langle a \rangle (p' \,;\, q) = (\langle a \rangle p') \,;\, q = (p \,;\, q)$"

$$\exists\, P, Q, p, q \cdot \langle a \rangle t = (p \,;\, q)\ \wedge\ PP \xrightarrow{p} P\ \wedge\ QQ \xrightarrow{q} Q$$
$$\wedge\ R = COND(last(p) = \checkmark, (Q \,;\, P), P)$$

From (B.4)

$$\exists\, P, QQ' \cdot PP \xrightarrow{\checkmark} P\ \wedge\ QQ \xrightarrow{a} QQ'\ \wedge\ \langle QQ', P \rangle \xrightarrow{t} R$$

$=$ "Using Lemma 4.8"

$$\exists\, P, QQ' \cdot PP \xrightarrow{\checkmark} P\ \wedge\ QQ \xrightarrow{a} QQ'\ \wedge\ \exists\, Q \cdot QQ' \xrightarrow{t} Q\ \wedge\ R = (Q \,;\, P)$$

$=$ "Removing QQ' "

$$\exists\, P, Q \cdot PP \xrightarrow{\checkmark} P\ \wedge\ QQ \xrightarrow{\langle a \rangle t} Q\ \wedge\ R = (Q \,;\, P)$$

$=\ \exists\, P, Q, p, q \cdot p = \langle \checkmark \rangle\ \wedge\ q = \langle a \rangle t\ \wedge\ \langle a \rangle t = (p \,;\, q)$
$$\wedge\ PP \xrightarrow{p} P\ \wedge\ QQ \xrightarrow{q} Q\ \wedge\ R = (Q \,;\, P)$$

B.2   Proof of Lemma 4.9

$$(P \rhd Q) \xrightarrow{t} 0 = \exists\, p, q \cdot t = (p \rhd q) \wedge P \xrightarrow{p} 0 \wedge Q \xrightarrow{q} 0$$

Basic Step: Case: $\langle \omega \rangle$

$$(P \rhd Q) \xrightarrow{\langle \omega \rangle} 0$$

$= (P \rhd Q) \xrightarrow{\omega} 0$

$=$ "From derived equation 4.14"

$$P \xrightarrow{!} 0 \wedge Q \xrightarrow{\omega} 0 \tag{B.5}$$
$$\vee\ P \xrightarrow{\omega} 0 \wedge \omega \neq \,! \tag{B.6}$$

From (B.5)

$$P \xrightarrow{\;!\;} 0 \wedge Q \xrightarrow{\;\omega\;} 0$$

$$= \exists\, p, q \cdot p = \langle\,!\,\rangle \wedge q = \langle \omega \rangle \wedge P \xrightarrow{\;p\;} 0 \wedge Q \xrightarrow{\;q\;} 0$$

$$= \text{``By using trace rule: } (p \rhd q) = (\langle\,!\,\rangle \rhd \langle \omega \rangle) = \langle \omega \rangle\text{''}$$

$$\exists\, p, q \cdot p = \langle\,!\,\rangle \wedge \langle \omega \rangle = (p \rhd q) \wedge P \xrightarrow{\;p\;} 0 \wedge Q \xrightarrow{\;q\;} 0$$

From (B.6)

$$P \xrightarrow{\;\omega\;} 0 \wedge \omega \neq\; !$$

$$= \exists\, p \cdot p = \langle \omega \rangle \wedge \omega \neq\; ! \wedge P \xrightarrow{\;p\;} 0$$

$$= \text{``By using theorem 4.4''}$$

$$\exists\, p \cdot p = \langle \omega \rangle \wedge \omega \neq\; ! \wedge P \xrightarrow{\;p\;} 0 \wedge \exists\, q \cdot Q \xrightarrow{\;q\;} 0$$

$$= \exists\, p, q \cdot p = \langle \omega \rangle \wedge \omega \neq\; ! \wedge P \xrightarrow{\;p\;} 0 \wedge Q \xrightarrow{\;q\;} 0$$

$$= \text{``By using trace rule: } \langle \omega \rangle \rhd q = \langle \omega \rangle \text{ where } \omega \neq\; !\,\text{''}$$

$$\exists\, p, q \cdot p = \langle \omega \rangle \wedge \omega \neq\; ! \wedge \langle \omega \rangle = (p \rhd q) \wedge P \xrightarrow{\;p\;} 0 \wedge Q \xrightarrow{\;q\;} 0$$

Combining the above two derivations

$$\exists\, p, q \cdot p = \langle\,!\,\rangle \wedge \langle \omega \rangle = (p \rhd q) \wedge P \xrightarrow{\;p\;} 0 \wedge Q \xrightarrow{\;q\;} 0$$

$$\vee\; \exists\, p, q \cdot p = \langle \omega \rangle \wedge \omega \neq\; ! \wedge \langle \omega \rangle = (p \rhd q) \wedge P \xrightarrow{\;p\;} 0 \wedge Q \xrightarrow{\;q\;} 0$$

$$= \exists\, p, q \cdot \langle \omega \rangle = (p \rhd q) \wedge P \xrightarrow{\;p\;} 0 \wedge Q \xrightarrow{\;q\;} 0$$

Inductive Step: Case: $\langle a \rangle t$

$$(P \rhd Q) \xrightarrow{\;\langle a \rangle t\;} 0$$

$$= \exists\, R \cdot (P \rhd Q) \xrightarrow{\;a\;} R \wedge R \xrightarrow{\;t\;} 0$$

$$= \text{``From derived equation 4.15''}$$

$$\exists\, P' \cdot P \xrightarrow{\;a\;} P' \wedge (P' \rhd Q) \xrightarrow{\;t\;} 0 \tag{B.7}$$

$$\vee\; P \xrightarrow{\;!\;} 0 \wedge Q \xrightarrow{\;\langle a \rangle t\;} 0 \tag{B.8}$$

From (B.7)

$$\exists\, P' \cdot P \xrightarrow{a} P' \wedge (P' \rhd Q) \xrightarrow{t} 0$$

= "Inductive hypothesis"

$$\exists\, P' \cdot P \xrightarrow{a} P' \wedge \exists\, p', q \cdot t = (p' \rhd q) \wedge P' \xrightarrow{p'} 0 \wedge Q \xrightarrow{q} 0$$

= "Removing P' "

$$\exists\, p', q \cdot t = (p' \rhd q) \wedge P \xrightarrow{\langle a \rangle p'} 0 \wedge Q \xrightarrow{q} 0$$

= "By trace rule: $\langle a \rangle t = \langle a \rangle (p' \rhd q) = (\langle a \rangle p' \rhd q)$ "

$$\exists\, p' q \cdot \langle a \rangle t = (\langle a \rangle p' \rhd q) \wedge P \xrightarrow{\langle a \rangle p'} 0 \wedge Q \xrightarrow{q} 0$$

$$= \exists\, p, p', q \cdot p = \langle a \rangle p' \wedge \langle a \rangle t = (p \rhd q) \wedge P \xrightarrow{p} 0 \wedge Q \xrightarrow{q} 0$$

$$= \exists\, p, q \cdot \langle a \rangle t = (p \rhd q) \wedge P \xrightarrow{p} 0 \wedge Q \xrightarrow{q} 0$$

From (B.8)

$$P \xrightarrow{!} 0 \wedge Q \xrightarrow{\langle a \rangle t} 0$$

$$= \exists\, p, q \cdot p = \langle ! \rangle \wedge q = \langle a \rangle t \wedge P \xrightarrow{p} 0 \wedge Q \xrightarrow{q} 0$$

= "By trace rule: $(\langle ! \rangle \rhd q) = q$ "

$$\exists\, p, q \cdot p = \langle ! \rangle \wedge \langle a \rangle t = (p \rhd q) \wedge P \xrightarrow{p} 0 \wedge Q \xrightarrow{q} 0$$

$$= \exists\, p, q \cdot \langle a \rangle t = (p \rhd q) \wedge P \xrightarrow{p} 0 \wedge Q \xrightarrow{q} 0$$

B.3   Proof of Lemma 4.10

$$P \parallel Q \xrightarrow{t} 0 \;=\; \exists\, p, q \cdot t \in (p \parallel q) \wedge P \xrightarrow{p} 0 \wedge Q \xrightarrow{q} 0$$

Basic step: $t = \langle \omega \rangle$

$$P \parallel Q \xrightarrow{\langle \omega \rangle} 0$$

$$= P \parallel Q \xrightarrow{\omega} 0$$

= "From operational rule 3.11"

$$\exists\, \omega 1, \omega 2 \cdot P \xrightarrow{\langle \omega 1 \rangle} 0 \wedge Q \xrightarrow{\langle \omega 2 \rangle} 0 \wedge \omega = \omega 1 \& \omega 2 \wedge \omega 1 \& \omega 2 \in (\langle \omega 1 \rangle \parallel \langle \omega 2 \rangle)$$

$$= \exists\, p, q \cdot p = \langle \omega 1 \rangle \wedge q = \langle \omega 2 \rangle \wedge \langle \omega \rangle \in (p \parallel q) \wedge P \xrightarrow{p} 0 \wedge Q \xrightarrow{q} 0$$

$$= \exists\, p, q \cdot \langle \omega \rangle \in (p \parallel q) \wedge P \xrightarrow{p} 0 \wedge Q \xrightarrow{q} 0$$

B.4   Proof of Lemma 4.11

$$PP \parallel QQ \xrightarrow{t} R$$
$$= \exists\, P, Q, p, q \cdot t \in (p \parallel q) \wedge PP \xrightarrow{p} P \wedge QQ \xrightarrow{q} P \wedge R = P \parallel Q$$

Proof:

Basic step: $t = \langle \omega \rangle$

$$PP \parallel QQ \xrightarrow{\langle \omega \rangle} R$$

$= PP \parallel QQ \xrightarrow{\omega} R$

$=$ "From derived equation 4.20"

$\exists P, Q \cdot \omega = \omega 1 \& \omega 2 \ \wedge\ \omega 1 \& \omega 2 \in (\langle \omega 1 \rangle \parallel \langle \omega 2 \rangle)$

$\wedge\ PP \xrightarrow{\omega 1} P \ \wedge\ QQ \xrightarrow{\omega 2} Q \ \wedge\ R = (P \parallel Q)$

$=\ \exists P, Q, p, q \cdot p = \langle \omega 1 \rangle \ \wedge\ q = \langle \omega 2 \rangle \ \wedge\ \langle \omega \rangle \in (p \parallel q)$

$\wedge\ PP \xrightarrow{p} P \ \wedge\ QQ \xrightarrow{q} Q \ \wedge\ R = (P \parallel Q)$

$=\ \exists P, Q, p, q \cdot \langle \omega \rangle \in (p \parallel q) \ \wedge\ PP \xrightarrow{p} P \ \wedge\ QQ \xrightarrow{q} Q \ \wedge\ R = (P \parallel Q)$

Inductive step: $t = \langle a \rangle t$

$$PP \parallel QQ \xrightarrow{\langle a \rangle t} R$$

$=\ \exists RR \cdot PP \parallel QQ \xrightarrow{a} RR \ \wedge\ RR \xrightarrow{t} R$

$=$ "From derived equation 4.21"

$\exists PP' \cdot PP \xrightarrow{a} PP' \ \wedge\ PP' \parallel QQ \xrightarrow{t} R$

$\vee\ \exists QQ' \cdot QQ \xrightarrow{a} QQ' \ \wedge\ PP \parallel QQ' \xrightarrow{t} R$

$=$ "Inductive hypothesis"

$\exists PP' \cdot PP \xrightarrow{a} PP' \ \wedge\ \exists P, Q, p', q \cdot t \in (p' \parallel q)$

$\wedge\ PP' \xrightarrow{p'} P \ \wedge\ QQ \xrightarrow{q} Q \ \wedge\ R = (P \parallel Q)$

$\vee\ \exists QQ' \cdot QQ \xrightarrow{a} QQ' \ \wedge\ \exists P, Q, p, q' \cdot t \in (p \parallel q')$

$\wedge\ PP \xrightarrow{p} P \ \wedge\ QQ \xrightarrow{q'} Q \ \wedge\ R = (P \parallel Q)$

$=$ "Removing PP', QQ' "

$\exists P, Q, p', q \cdot t \in (p' \parallel q) \ \wedge\ PP \xrightarrow{\langle a \rangle p'} P \ \wedge\ QQ \xrightarrow{q} Q \ \wedge\ R = (P \parallel Q)$

$\vee\ \exists P, Q, p, q' \cdot t \in (p \parallel q') \ \wedge\ PP \xrightarrow{p} P \ \wedge\ QQ \xrightarrow{\langle a \rangle q'} Q \ \wedge\ R = (P \parallel Q)$

$=\ \exists P, Q, p, q \cdot p = \langle a \rangle p' \ \wedge\ t \in (p' \parallel q) \ \wedge\ PP \xrightarrow{p} P \ \wedge\ QQ \xrightarrow{q} Q \ \wedge\ R = (P \parallel Q)$

$\vee\ \exists P, Q, p, q \cdot q = \langle a \rangle q' \ \wedge\ t \in (p \parallel q') \ \wedge\ PP \xrightarrow{p} P \ \wedge\ QQ \xrightarrow{q} Q \ \wedge\ R = (P \parallel Q)$

$=$ "Combining existential quantifications"

$\exists P, Q, p, q \cdot (p = \langle a \rangle p' \ \wedge\ t \in (p' \parallel q) \ \vee\ q = \langle a \rangle q' \ \wedge\ t \in (p \parallel q'))$

$\wedge\ PP \xrightarrow{p} P \ \wedge\ QQ \xrightarrow{q} Q \ \wedge\ R = (P \parallel Q)$

$=$ "By the definition of traces"

$\exists P, Q, p, q \cdot \langle a \rangle t \in (p \parallel q) \ \wedge\ PP \xrightarrow{p} P \ \wedge\ QQ \xrightarrow{q} Q \ \wedge\ R = (P \parallel Q)$

B.5   Proof of Lemma 4.12

$$(P \div Q) \xrightarrow{(t,t')} 0 \;=\; \exists\, p, q \cdot (t, t') = (p \div q) \;\wedge\; P \xrightarrow{p} 0 \;\wedge\; Q \xrightarrow{q} 0$$

Proof:

Basic step: $t = \langle \omega \rangle$

$$(P \div Q) \xrightarrow{(\omega, t')} 0$$

$$= \; \exists\, R \cdot (P \div Q) \xrightarrow{\omega} R \;\wedge\; R \xrightarrow{t'} 0$$

$= \;$ "From derived equation 4.22"

$$P \xrightarrow{\checkmark} 0 \;\wedge\; Q \xrightarrow{t'} 0 \tag{B.9}$$

$$\vee \; \exists\, R \cdot P \xrightarrow{\omega} 0 \;\wedge\; \omega \neq \checkmark \;\wedge\; R = SKIP \;\wedge\; SKIP \xrightarrow{t'} 0 \;\wedge\; t' = \langle \checkmark \rangle \tag{B.10}$$

From equation B.9

$$P \xrightarrow{\checkmark} 0 \;\wedge\; Q \xrightarrow{t'} 0$$

$$= \; \exists\, p, q \cdot p = \langle \checkmark \rangle \;\wedge\; q = t' \;\wedge\; P \xrightarrow{p} 0 \;\wedge\; Q \xrightarrow{q} 0$$

$= \;$ "By trace rule: $(\langle \checkmark \rangle, t') = (p \div q)$"

$$\exists\, p, q \cdot (\langle \omega \rangle, t') = (p \div q) \;\wedge\; \omega = \langle \checkmark \rangle \;\wedge\; P \xrightarrow{p} 0 \;\wedge\; Q \xrightarrow{q} 0$$

From equation B.10

$$\exists\, R \cdot P \xrightarrow{\omega} 0 \;\wedge\; \omega \neq \checkmark \;\wedge\; R = SKIP \;\wedge\; SKIP \xrightarrow{t'} 0 \;\wedge\; t' = \langle \checkmark \rangle$$

$$= \; P \xrightarrow{\omega} 0 \;\wedge\; \omega \neq \checkmark \;\wedge\; SKIP \xrightarrow{\checkmark} 0$$

$$= \; \exists\, p, q \cdot p = \langle \omega \rangle \;\wedge\; \omega \neq \checkmark \;\wedge\; q = \langle \checkmark \rangle \;\wedge\; P \xrightarrow{p} 0 \;\wedge\; SKIP \xrightarrow{q} 0$$

$$= \; \exists\, Q, p, q \cdot p = \langle \omega \rangle \;\wedge\; q = \langle \checkmark \rangle \;\wedge\; Q = SKIP \;\wedge\; P \xrightarrow{p} 0 \;\wedge\; Q \xrightarrow{q} 0$$

$= \;$ "By using trace rule"

$$\exists\, p, q \cdot (\langle \omega \rangle, \langle \checkmark \rangle) = (p \div q) \;\wedge\; \omega \neq \checkmark \;\wedge\; P \xrightarrow{p} 0 \;\wedge\; Q \xrightarrow{q} 0$$

Inductive step: $t = \langle a \rangle t$

$$(P \div Q) \xrightarrow{\langle a \rangle t, t'} 0$$

$$= \quad \exists\, RR \cdot (P \div Q) \xrightarrow{a} RR \ \wedge\ RR \xrightarrow{t, t'} 0$$

$$= \quad \text{``From derived equation 4.23''}$$

$$\exists\, P' \cdot P \xrightarrow{a} P' \ \wedge\ (P' \div Q) \xrightarrow{t, t'} 0$$

$$= \quad \text{``Inductive hypothesis''}$$

$$\exists\, P' \cdot P \xrightarrow{a} P' \ \wedge\ \exists\, p', q \cdot (t, t') = (p' \div q) \ \wedge\ P' \xrightarrow{p'} 0 \ \wedge\ Q \xrightarrow{q} 0$$

$$= \quad \text{``Removing } P' \text{ ''}$$

$$\exists\, p', q \cdot (t, t') = (p' \div q) \ \wedge\ P \xrightarrow{\langle a \rangle p'} 0 \ \wedge\ Q \xrightarrow{q} 0$$

$$= \quad \exists\, p, p', q \cdot p = \langle a \rangle p' \ \wedge\ (t, t') = (p' \div q) \ \wedge\ P \xrightarrow{p} 0 \ \wedge\ Q \xrightarrow{q} 0$$

$$= \quad \text{``By trace rule: } (t, t') = (p' \div q) \Rightarrow (\langle a \rangle t, t') = (\langle a \rangle p' \div q) = (p \div q) \text{''}$$

$$\exists\, p, q \cdot (\langle a \rangle t, t') = (p \div q) \ \wedge\ P \xrightarrow{p} 0 \ \wedge\ Q \xrightarrow{q} 0$$

Deriving correspondence:

$$(t, t') \in DT(P \div Q)$$

$$= (P \div Q) \xrightarrow{(t, t')} 0$$

$$= \quad \text{``From Lemma 4.12''}$$

$$\exists\, p, q \cdot (t, t') = (p \div q) \ \wedge\ P \xrightarrow{p} 0 \ \wedge\ Q \xrightarrow{q} 0$$

$$= \quad \text{``By trace derivation rules''}$$

$$\exists\, p, q \cdot (t, t') = (p \div q) \ \wedge\ p \in DT(P) \ \wedge\ q \in DT(Q)$$

$$= \quad \text{``Structural induction''}$$

$$\exists\, p, q \cdot (t, t') = (p \div q) \ \wedge\ p \in T(P) \ \wedge\ q \in T(Q)$$

$$= \quad \text{``By trace rule''}$$

$$= (t, t') \in T(P \div Q) \qquad\qquad\qquad\qquad\qquad\qquad \square$$

Appendix C

PVS Proof Trees

In this appendix we present proof trees of some lemmas defined in Chapter 6. Proof trees give us a clear view of how the proofs have been carried out and show the commands that have been used in the proofs. In PVS it is possible to explore each node of the proof tree. The proof trees for the lemmas of the standard sequential and parallel composition, the transaction block, and the compensation pair, and the lemmas for the synchronised composition of both standard, and compensable processes are presented in the following sections.

C.1   Standard Sequential Composition
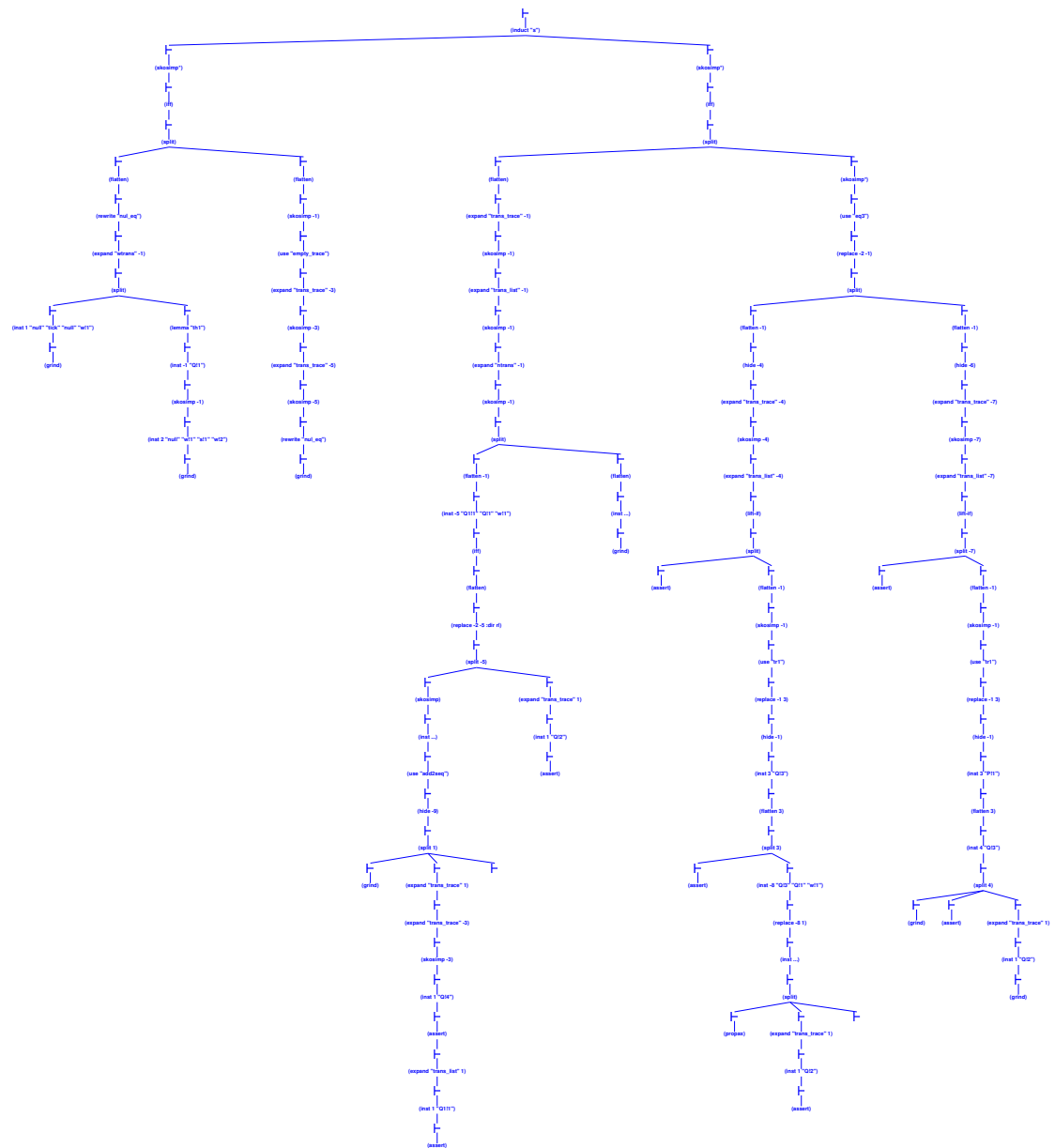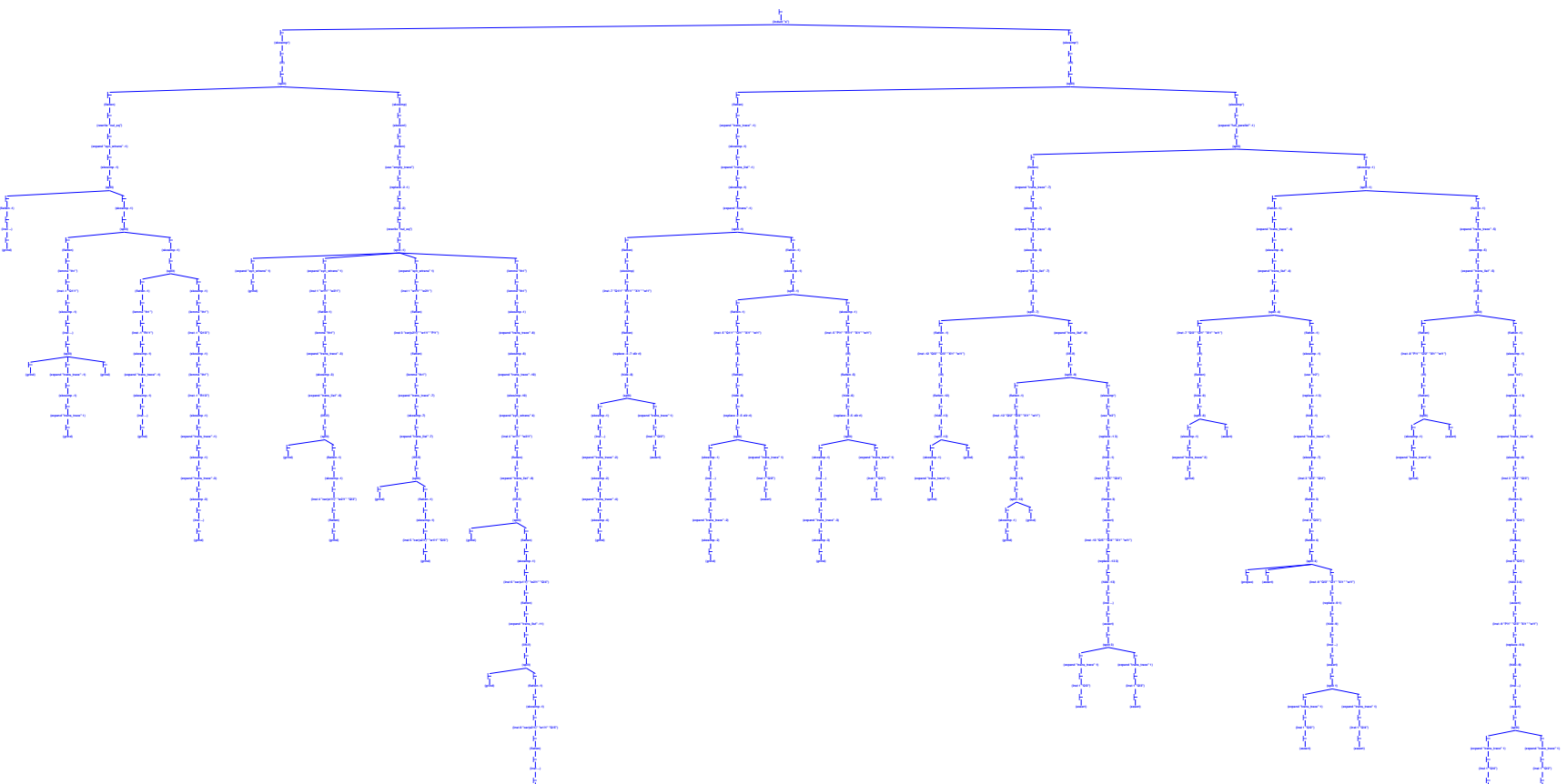
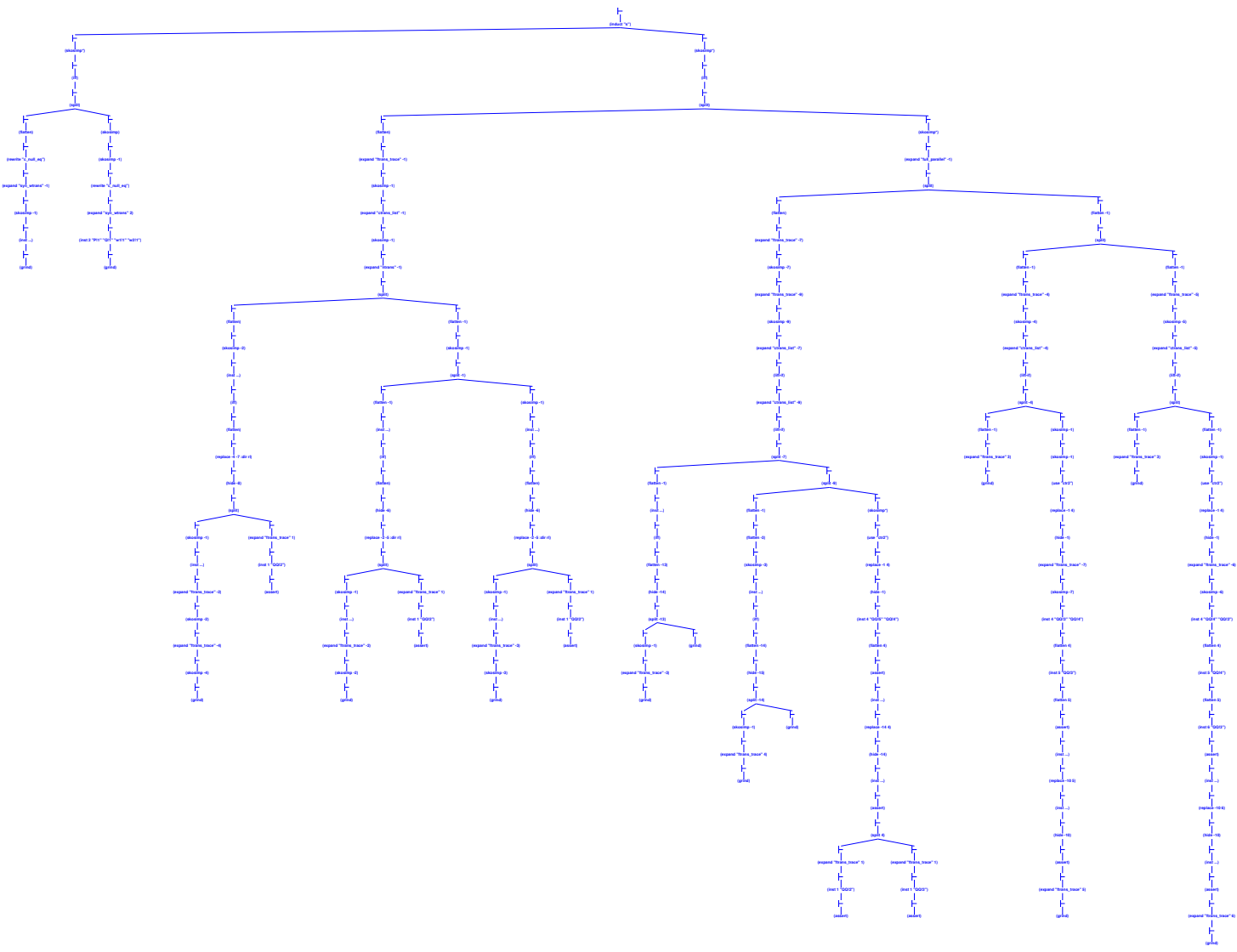## C.2 Standard Parallel Composition
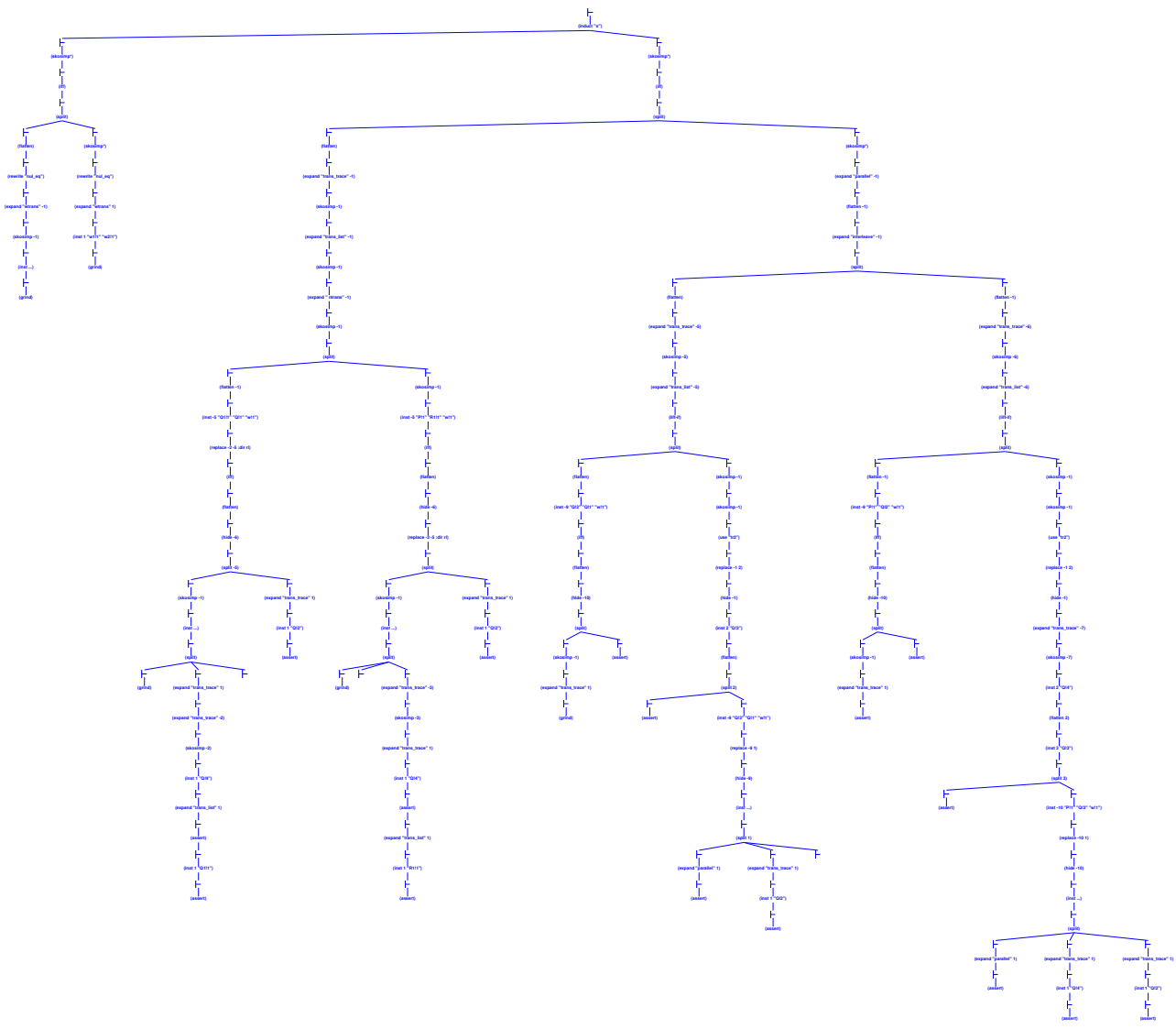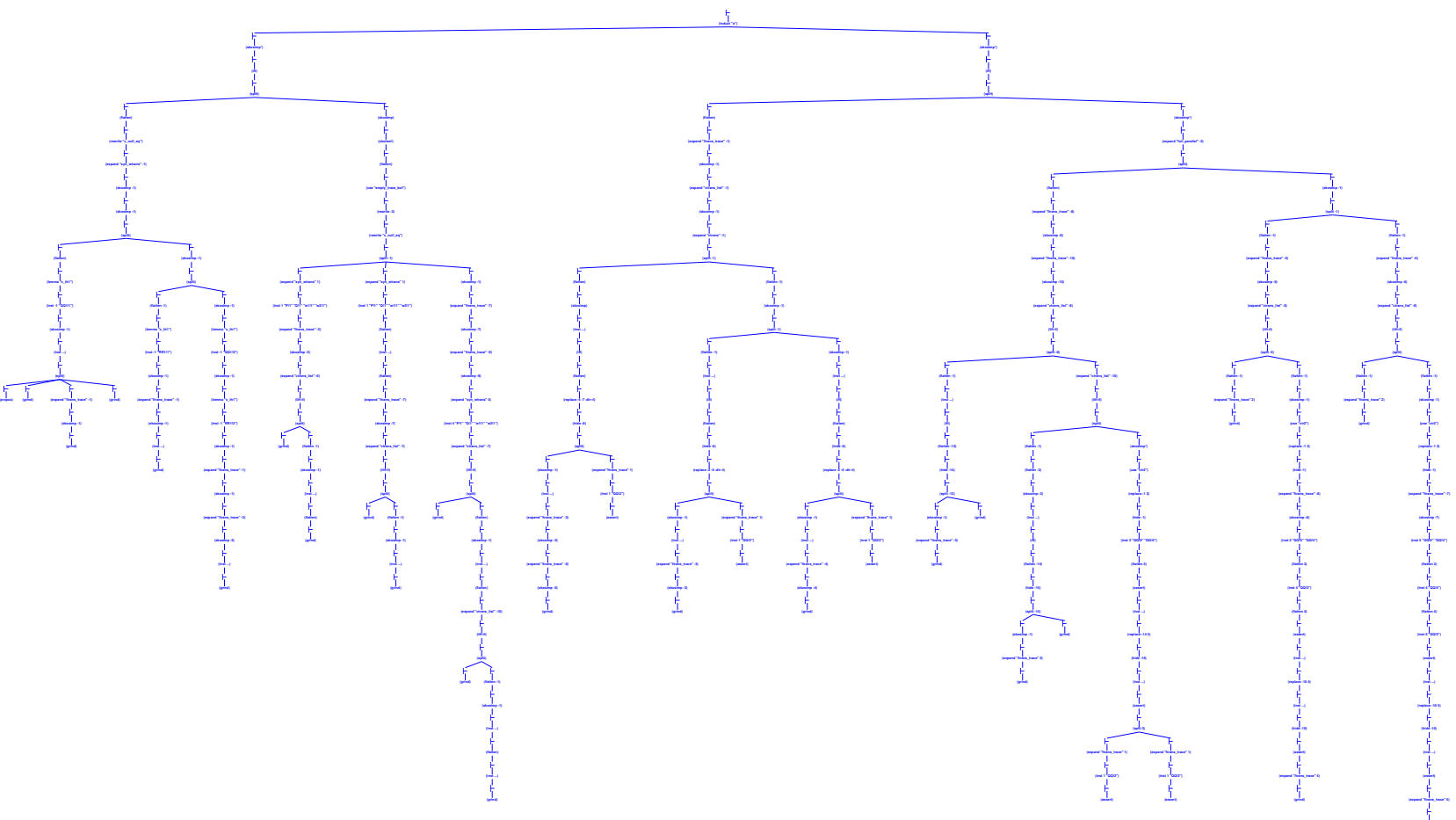
C.3 Transaction Block
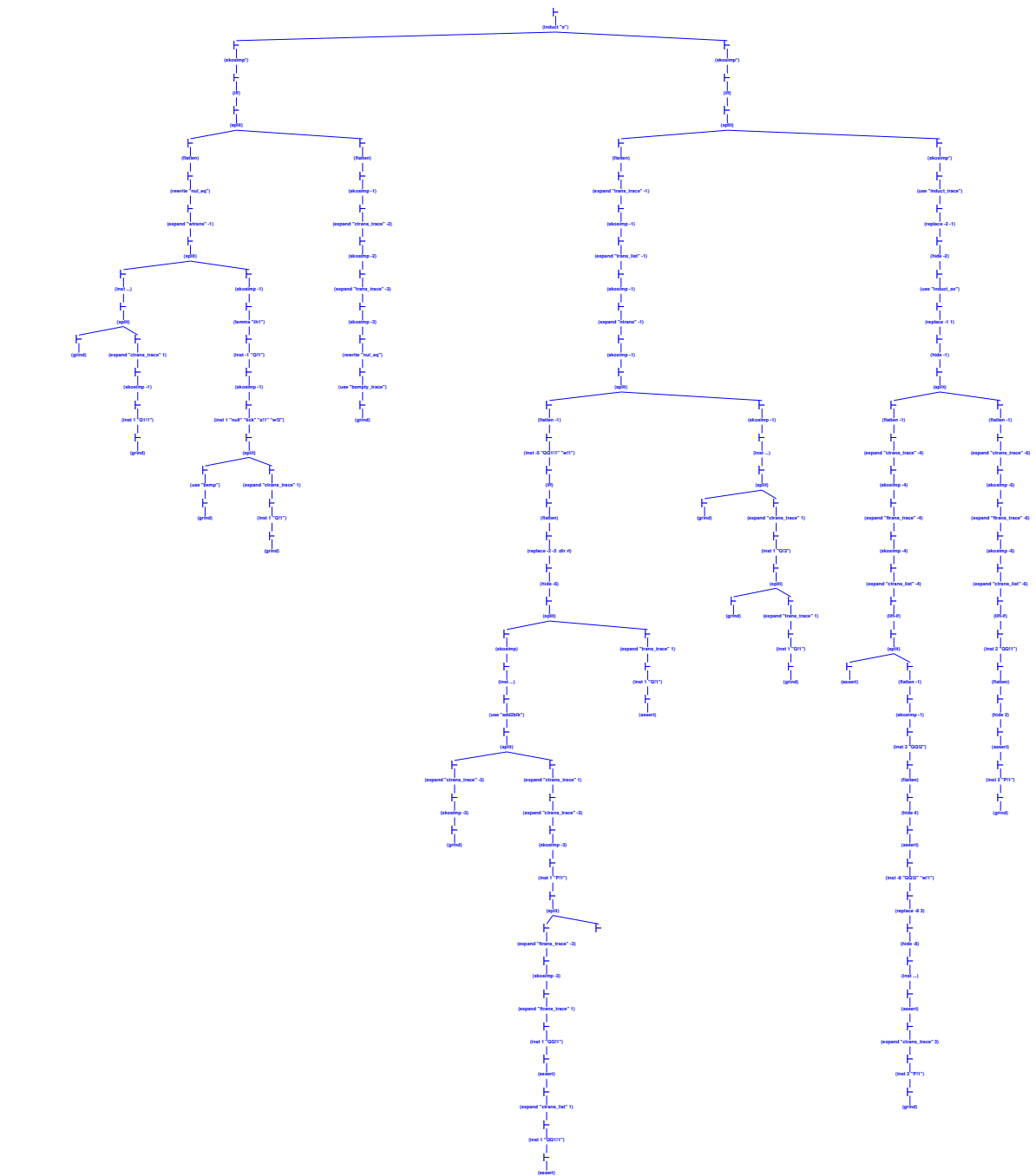
C.4   Compensation Pair
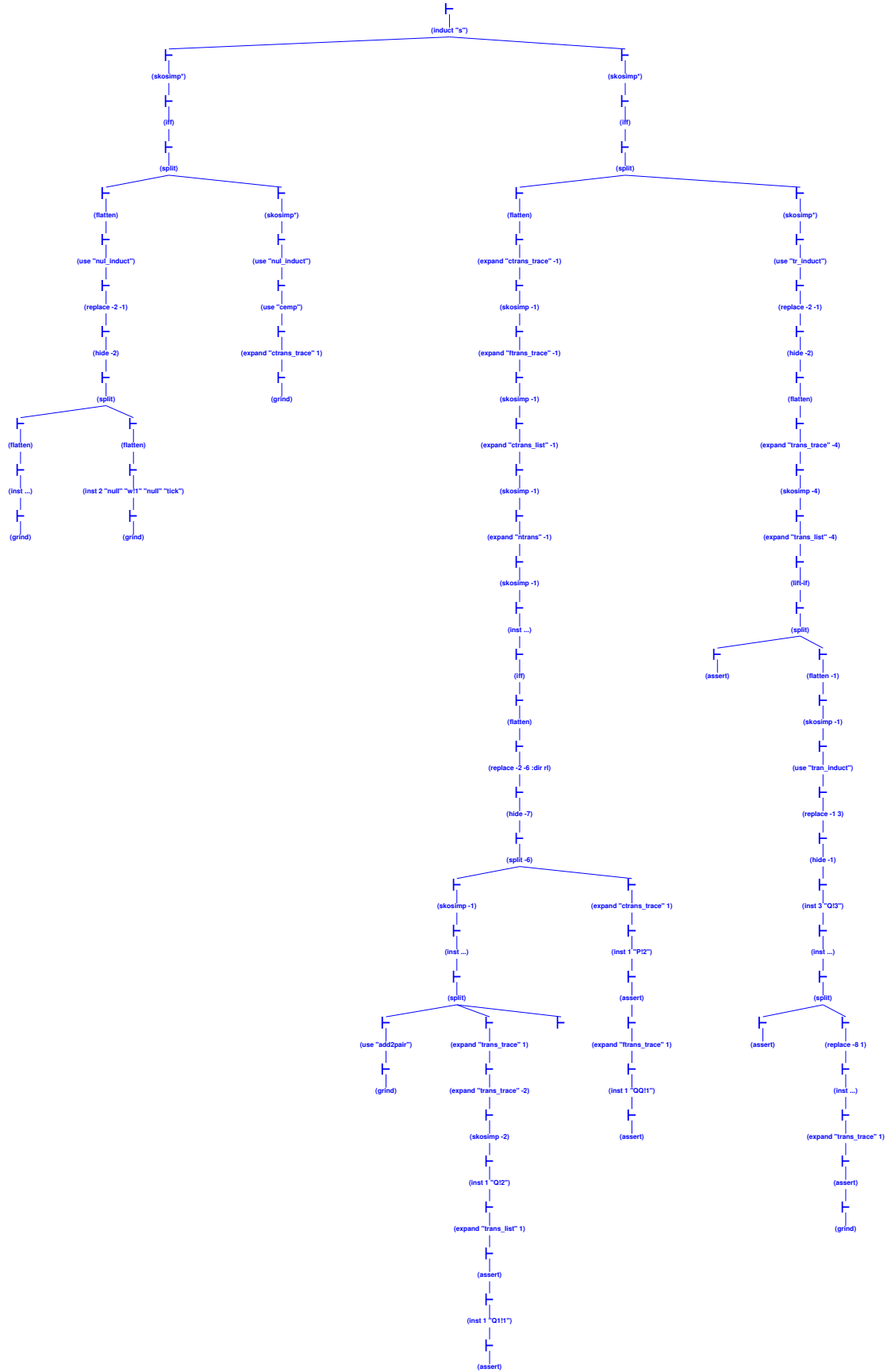
C.5 Standard Synchronised Parallel Composition

C.7 Compensable Synchronised Parallel Composition (with bottom)

Bibliography

[1] Business process modeling language (BPML). [www.bpmi.org].

[2] WSCI Specifications. [http://www.w3.org/TR/wsci].

[3] J. R Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[4] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, and Laurent Voisin. An open extensible tool environment for event-b. In Zhiming Liu and Jifeng He, editors, *ICFEM*, volume 4260 of *LNCS*, pages 588–605. Springer, 2006.

[5] Jean-Raymond Abrial and Dominique Cansell. Click'n prove: Interactive proofs within set theory. In David A. Basin and Burkhart Wolff, editors, *TPHOLs*, volume 2758 of *LNCS*, pages 1–24. Springer, 2003.

[6] L. Aceto, W. Fokkink, and C. Verhoef. *Structural operational semantics.*, chapter 3, pages 197–291. Handbook of Process Algebra. Elsevier, 2001.

[7] Myla Archer and Constance L. Heitmeyer. Human-style Theorem Proving Using PVS. In Elsa L. Gunter and Amy P. Felty, editors, *Theorem Proving in Higher Order Logics, 10th International Conference, TPHOLs'97*, volume 1275 of *LNCS*, pages 33–48, 1997.

[8] Juan C. Augusto, Michael Leuschel, Michael Butler, and Carla Ferreira. Using the extensible model checker XTL to verify StAC business specifications. In *3rd Workshop on Automated Verification of Critical Systems (AVoCS 2003)*, pages 253–266, Southampton, UK, 2003.

[9] J. C. M. Baeten. A brief history of process algebra. *Theoretical Computer Science*, 335(2-3):131–146, 2005.

[10] J. C. M. Baeten and W. P. Weijland. *Process Algebra*. Number 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[11] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicael Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan

Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saibi, and Benjamin Werner. The Coq proof assistant reference manual : Version 6.1. Technical Report 0203, INRIA, August 1997.

[12] Twan Basten and Jozef Hooman. Process Algebra in PVS. In Rance Cleaveland, editor, *TACAS'99*, volume 1579 of *LNCS*, pages 270–284. Springer-Verlag, 1999.

[13] Martin Berger and Kohei Honda. The two-phase commitment protocol in an extended pi-calculus. *Electronic Notes in Theoretical Computer Science*, 39(1), 2000.

[14] J.A. Bergstra, A. Ponse, and S.A. Smolka, editors. *Handbook of Process Algebra*. Elsevier, 2001.

[15] Andrew P. Black, Vincent Cremet, Rachid Guerraoui, and Martin Odersky. An equational theory for transactions. In Paritosh K. Pandya and Jaikumar Radhakrishnan, editors, *FSTTCS 2003: Foundations of Software Technology and Theoretical Computer Science*, volume 2914 of *LNCS*, pages 38–49, Mumbai, India, December 15-17 2003. Springer-Verlag.

[16] Laura Bocchi, Cosimo Laneve, and Gianluigi Zavattaro. A calulus for long-running transactions. In *FMOODS'03*, volume 2884 of *LNCS*, pages 124–138. Springer-Verlag, 2003.

[17] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Network and ISDN Systems*, 14(1):25–59, 1987.

[18] Antonio Brogi, Carlos Canal, Ernesto Pimentel, and Antonio Vallecillo. Formalizing web service choreographies. In *WS-FM' 04, ENTCS*, volume 105, pages 73–94. Elsevier, 2004.

[19] Roberto Bruni, Michael Butler, Carla Ferreira, Tony Hoare, Hernan Melgratti, and Ugo Montanari. Comparing two approaches to compensable flow composition. In Martn Abadi and Luca de Alfaro, editors, *CONCUR 2005*, volume 3653 of *LNCS*, pages 383–397, 2005.

[20] Roberto Bruni, Cosimo Laneve, and Ugo Montanari. Orchestrating transactions in join calculus. In Lubos Brim, Petr Jancar, Mojmír Kretínský, and Antonín Kucera, editors, *CONCUR '02: Proceedings of the 13th International Conference on Concurrency Theory*, volume 2421 of *LNCS*, pages 321–337, 2002.

[21] Roberto Bruni, Hernán Melgratti, and Ugo Montanari. Theoretical foundations for compensations in flow composition languages. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT sysposium on Principles of programming languages*, pages 209–220. ACM Press, 2005.

[22] Nadia Busi, Roberto Gorrieri, Claudio Guidi, Roberto Lucchi, and Gianluigi Za-
vattaro. Towards a formal framework for choreography. In *International Workshop
on Distributed and Mobile Collaboration (DMC 2005)*,. IEEE Computer Society
Press, 2005.

[23] Michael Butler. csp2B: A practical approach to combining csp and B. *Formal
Aspects of Computing*, 12:182–196, 2000.

[24] Michael Butler and Carla Ferreira. A process compensation language. In *Integrated
Formal Methods(IFM'2000)*, volume 1945 of *LNCS*, pages 61 – 76. Springer-Verlag,
2000.

[25] Michael Butler and Carla Ferreira. An operational semantics for StAC, a language
for modelling long-running business transactions. In *Coordination 2004*, volume
2949 of *LNCS*. Springer-Verlag, 2004.

[26] Michael Butler and Tony Hoare. Towards refinement of joint transaction. [Draft].

[27] Michael Butler, Tony Hoare, and Carla Ferreira. A trace semactics for long-running
transaction. In A.E. Abdallah, C.B. Jones, and J.E. Sanders, editors, *Proceedings
of 25 Years of CSP*, volume 3525 of *LNCS*, London, 2004. Springer-Verlag.

[28] Michael Butler and Michael Leuschel. Combining CSP and B for specification
and property verification. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki,
editors, *FM – 2005*, volume 3582 of *LNCS*, pages 221–236. Springer-Verlag, 2005.

[29] Michael Butler and Shamim Ripon. Executable semantics for compensating CSP.
In Mario Bravetti, Leïla Kloul, and Gianluigi Zavattaro, editors, *WS-FM 2005*,
volume 3670 of *LNCS*, pages 243–256. Springer-Verlag, September 1-3 2005.

[30] Albert John Camilleri. Mechanizing CSP trace theory in High Order Logic. *IEEE
Transactions on Software Engineering*, 16(9):993–1004, September 1990.

[31] M. Chessell, C. Griffin, D. Vines, Michael Butler, Carla Ferreira, and P. Hender-
son. Extending the concept of transaction compensation. *IBM Systems Journal*,
41(4):743–758, 2002.

[32] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services
description language (WSDL) 1.1. W3c note, World Wide Web Consortium, March
2001.

[33] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-
state concurrent systems using temporal logic specifications. *ACM Transactions
on Programming Languages and Systems*, 8(2):224–263, 1986.

[34] E. M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press,
1999.

[35] F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. *Business process execution language for web services, version 1.1.*, 2003. [http://www-106.ibm.com/developerworks/library/ws-bpel/].

[36] Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana. Unraveling the web services web: An introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6:86–93, 2002.

[37] Vincent Danos and Jean Krivine. Transactions in RCCS. In Martn Abadi and Luca de Alfaro, editors, *CONCUR 2005*, volume 3653 of *LNCS*, pages 398– 412, 2005.

[38] Bruno Dutertre and Steve Schneider. Using a PVS embedding of CSP to verify authentication protocols. In Elsa L. Gunter and Amy P. Felty, editors, *Theorem Proving in Higher Order Logics, 10th International Conference, TPHOLs'97*, volume 1275 of *LNCS*, pages 121–136. Springer-Verlag, 1997.

[39] Neil Evans. *Investigating Security Through Proof.* PhD thesis, Royal Holloway, University of London, 2003.

[40] Neil Evans and Steve A. Schneider. Verifying security protocols with PVS: widening the rank function approach. *Journal of Logic and Algebraic Programming*, 64(2):253–284, August 2005.

[41] Andrea Ferrara. Web Services: a Process Algebra Approach. In Marco Aiello, Mikio Aoyama, Francisco Curbera, and Mike P. Papazoglou, editors, *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 242–251, New York, NY, USA, November 2004. ACM Press.

[42] Carla Ferreira. *Process Modelling of Business Processes with Compensation.* PhD thesis, Unversity of Southampton, November 2002.

[43] R. W. Floyd. Assigning Meanings to Programs. In *American Mathematical Society Symposium in Applied Mathematics.*, pages 19–31, 1967.

[44] Formal Systems (Europe) Ltd. *Failure-Divergences Refinement: FDR2 User Manual*, 1997.

[45] Formal Systems (Europe) Ltd. *Process Behaviour Explorer(PrOBE) User Manual*, January 2003.

[46] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the Join-calculus. In *POPL '96, 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 372–385. ACM Press, 1996.

[47] G. D. Plotkin. An operational semantics for CSP. In D. Bjørner, editor, *IFIP TC2 Working Conference on Formal Description of Programming Concepts -II*, pages 199–255. North-Holland, 1983.

[48] A. J. Galloway and W. J. Stoddart. An operational semantics for ZCCS. In M. G. Hinchey and S. Liu, editors, *1st International Conference of Formal Engineering Methods (ICFEM'97)*, pages 272–282, Japan, 1997. IEEE Computer Society Press.

[49] Hector Garcia-Molina and Kenneth Salem. Sagas. In Umeshwar Dayal and Irving L. Traiger, editors, *SIGMOD Conference*, pages 249–259. ACM Press, May 27-29 1987.

[50] M. J. C. Gordon. *The Denotational Description of Programming Languages, An Introduction.* Springer-Verlag, 1979.

[51] Mike Gordon. The semantic challenge of Verilog HDL. In *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science (LICS '95:)*, pages 136–145. IEEE Computer Society, June 1995.

[52] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[53] Jim Gray. The Transaction Concept: Virtues and Limitations (invited paper). In *Very Large Data Bases, 7th International Conference*, pages 144–154. IEEE Computer Society, 1981.

[54] Jim Gray and Andreas Reuter. *Transaction Processing : Concepts and Techniques.* Morgan Kaufmann Publishers, 1993.

[55] Rix Groenboom, Chris Hendriks, Indra Polak, Jan Terlouw, and Jan Tijmen Udding. Algebraic Proof Assistants in HOL. In *MPC '95: Mathematics of Program Construction*, volume 947 of *LNCS*, pages 304–321. Springer-Verlag, 1995.

[56] Matthew Hennessy. *The Semantics of Programming Languages: an Elementary Introduction using Structural Operational Semantics.* John Wiley & Sons, Inc., 1990.

[57] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[58] C. A. R. Hoare. Process algebra: A unifying approach. In Ali E. Abdallah, Cliff B. Jones, and Jeff W. Sanders, editors, *25 Years of CSP*, volume 3525 of *LNCS*, pages 36–60. Springer-Verlag, July 7-8 2004.

[59] C.A.R. Hoare. *Communicating Sequential Process.* Prentice Hall, 1985.

[60] C.A.R. Hoare and He Jifeng. *Unifying Theories of Programming.* Prentice Hall International Series in Computer Science, 1998.

[61] M. B. Josephs and J. T. Udding. An overview of DI algebra. In T. N. Mudge, V. Milutinovic, and L. Hunter, editors, *26th Hawaii Int. Conference on System Science (HICSS 1993)*, volume I, pages 329–338. IEEE Computer Society Press, 1993.

[62] Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *Proceedings of the Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 247 of *LNCS*, pages 22–39. Springer-Verlag, 1987.

[63] Cosimo Laneve and Gianluigi Zavattaro. Foundations of web transactions. In *FoSSaCS*, volume 3441 of *LNCS*, pages 282–298, 2005.

[64] Michael Leuschel. Design and implementation of the high level specification language CSP(LP) in Prolog. In I. V. Ramakrishnan, editor, *Proceedings of PADL'01*, volume 1990 of *LNCS*, pages 14–28. Springer-Verlag, March 2001.

[65] Michael Leuschel and Michael Butler. ProB: A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, September 2003.

[66] Frank Leymann. The web services flow language (WSFL 1.0). Technical report, Member IBM Academy of Technology, IBM Software Group, 2001. [http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf].

[67] Mark Little. Transactions and Web Services. *Communications of the ACM*, 46(10):49–54, October 2003.

[68] B. P. Mahony and J. S. Dong. Blending Object-Z and Timed CSP: An introduction to TCOZ. In *20th International Conference on Software Engineering (ICSE'98)*, pages 95–104, Japan, 1998. IEEE Computer Society Press.

[69] Manuel Mazzara and Roberto Lucchi. A framework for generic error handling in business processes. *Electronic Notes in Theoretical Computer Science*, 105:133–145, December 2004.

[70] John McCarthy. Towards a mathematical science of computation. In *IFIP Congress 62*, pages 21–28, 1962.

[71] T. F. Melham. A mechanized theory of the π-calculus in HOL. *Nordic Journal of Computing*, 1(1):50–76, 1994.

[72] L. G. Meredith and Steve Bjorg. Contracts and Types. *Communications of the ACM*, 46(10):41–47, October 2003.

[73] B. Metha, M. Levy, G. Meredith, T. Andrews, B. Beckman, J. Klein, and A. Mital. Biztalk server 2000 business process orchestration. *IEEE Data Engineering Bulletin,*, 24(1):35–39, 2001.

[74] J.-J. Ch. Meyer and E.P.de Vink. *On Relating Denotational and Operational Semantics for Programming Languages with Recursion and Concurrency*, chapter 24, pages 387–406. Open Problems in Topology. Elsevier, 1990.

[75] Nikola Milanovic and Miroslaw Malek. Current solutions for web service composition. *IEEE Internet Computing*, 8(6):51–59, 2004.

[76] Robin Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[77] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[78] Robin Milner. *Operational and Algebraic Semantics of Concurrent Processes.*, volume Volume B: Formal Models and Sematics of *Handbook of Theoretical Computer Science*, chapter 12, pages 1201–1242. Elsevier and MIT Press, 1990.

[79] Robin Milner. A calculus of mobile processes. *Journal of Information and computing*, 100(1):1–77, 1992.

[80] Peter D. Mosses. *Denotational semantics*, chapter 11, pages 575–631. MIT Press, 1990.

[81] Peter D. Mosses. Foundations of modular sos. In Miroslaw Kutylowski, Leszek Pacholski, and Tomasz Wierzbicki, editors, *Mathematical Foundations of Computer Science, 24th International Symposium, MFCS'99*, volume 1672 of *LNCS*, pages 70–80, Poland, September 6-10 1999. Springer-Verlag.

[82] Peter D. Mosses. The Varieties of Programming Language Semantics (Summary). In Jan van Leeuwen, Osamu Watanabe, Masami Hagiya, Peter D. Mosses, and Takayasu Ito, editors, *IFIP TCS*, volume 1872 of *LNCS*, pages 624–628. Springer-Verlag, 2000.

[83] Peter D. Mosses. The varieties of programming language semantics and their uses. In Dines Bjørner, Manfred Broy, and Alexandre V. Zamulin, editors, *PSI '02: Revised Papers from the 4th International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, volume 2244 of *LNCS*, pages 165–190. Springer-Verlag, 2001.

[84] Peter D. Mosses. Pragmatics of modular sos. In Hélène Kirchner and Christophe Ringeissen, editors, *Algebraic Methodology and Software Technology, 9th International Conference, AMAST 2002*, volume 2422 of *LNCS*, pages 21–40. Springer-Verlag, 2002.

[85] Peter D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61(195-228), 2004. Speical issue on SOS.

[86] Peter D. Mosses. Formal semantics of programming languages: - An overview -. *Electr. Notes Theor. Comput. Sci.*, 148(1):41–73, 2006.

[87] Hanne R. Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction.* John Wiley & Sons, Inc., 1992.

[88] OASIS. Introduction to UDDI: Important feature and functional concepts. Technical report, Organization for the Advancement of Structured Information Standard, 2004.

[89] S. Owre, J.M. Rushby, and N Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, June 1992.

[90] S. Owre, N Shankar, J.M. Rushby, and D.W.J.Stringrt-Calvert. *PVS Prover Guide.* SRI International, version 2.4 edition, November 2001.

[91] S. Owre, N Shankar, J.M. Rushby, and D.W.J.Stringrt-Calvert. *PVS System Guide.* SRI International, version 2.4 edition, November 2001.

[92] Sam Owre and Natarajan Shanker. Abstract datatypes in PVS. Technical Report SRI-CSL-93-9R, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993. Extensively revised June 1997.

[93] Joachim Parrow. *An Introduction to the π-Calculus*, chapter 8, Handbook of Process Algebra, pages 479–543. Handbook of Process Algebra. Elsevier, 2001.

[94] L.C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer-Verlag, 1994.

[95] Chris Peltz. Web services orchestration and choreography. *IEEE Computer*, 36(10):46–52, October 2003.

[96] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Computer Science Department, September 1981.

[97] Gordon D. Plotkin. A Structural Approach to Operational Semantics. *Journal of Logic and Algebraic Programming*, 60:17–139, 2004. This article first appeared as [96].

[98] Gordon D. Plotkin. The Origins of Structural Operational Semantics. *Journal of Logic and Algebraic Programming*, 60:3–15, 2004.

[99] Zongyan Qiu, Shuling Wang, Geguang Pu, and Xiangpeng Zhao. Semantics of BPEL4WS-like fault and compensation handling. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM 2005*, volume 3582 of *LNCS*, pages 350–365. Springer-Verlag, 2005.

[100] R. Boulton, A. Gordon, M.J.C. Gordon, J. Herbert, and J. van Tassel. Experience with embedding hardware description languages in HOL. In *Proc. of the International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 129–156. North-Holland, 1993.

[101] George M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58(1-3):249–261, June 1988.

[102] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, pearson edition, 1998.

[103] Gwen Salaün, Lucas Bordeaux, and Marco Schaerf. Describing and reasoning on web services using process algebra. In *Proceedings of the IEEE International Conference on Web Services (ICWS'04)*. IEEE Computer Society, June 6-9 2004.

[104] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., 1986.

[105] Steve Schneider. An operational semantics for timed CSP. *Journal of Information and computing*, 116(2):193–213, 1995.

[106] Steve Schneider, Jim Davies, D. M. Jackson, George M. Reed, Joy N. Reed, and A. W. Roscoe. Timed CSP: Theory and practice. In *REX Workshop*, volume 600 of *LNCS*, pages 640–675, 1991.

[107] Steve Schneider and Helen Treharne. Communicating B Machines. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *Proceedings of 2nd International Z and B Conference (ZB'02)*, volume 2272 of *LNCS*, pages 416–435, France, 2002. Springer-Verlag.

[108] Steve Schneider and Helen Treharne. Verifying Controlled Components. In E.A. Boiten, J. Derrick, and G. Smith, editors, *Proceedings of Integrated Formal Method (IFM 2004)*, volume 2999 of *LNCS*, pages 87–107, UK, 2004. Springer-Verlag.

[109] Dana Scott and Christopher Strachey. Toward a mathematical semantics for computer languages. Programming Research Group Technical Monograph PRG-6, Oxford Univ. Computing Lab., 1971.

[110] Natarajan Shankar and Sam Owre. Principles and Pragmatics of Subtyping in PVS. In Didier Bert, Christine Choppy, and Peter D. Mosses, editors, *Recent Trends in Algebraic Development Techniques, 14th International Workshop, WADT '99*, volume 1827 of *LNCS*, pages 37–52. Springer-Verlag, September 15-18 1999.

[111] Scott F. Smith. From operational to denotational semantics. In *Proceedings of the 7th International Conference on Mathematical Foundations of Programming Semantics*, volume 598 of *LNCS*, pages 54–76, 1992.

[112] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 2nd edition, 1992.

[113] S.Thatte. *XLANG: Web Services for Business Process Design.* Microsoft Corporation, 2001. [www.gotdotnet.com/team/xml/wsspace/xlang-c].

[114] Helen Treharne and Steve Schneider. Using Process Algebra to control B OPERATIONS. In K. Araki, A. Galloway, and K. Taguchi, editors, *1st International Conference on Integrated Formal Method (IFM'99)*, pages 437–457, UK, 1999. Springer-Verlag.

[115] Kenneth J. Turner. Formalising web services. In Farn Wang, editor, *Formal Techniques for Networked and Distributed Systems - FORTE 2005, 25th IFIP WG 6.1 International Conference*, volume 3731 of *LNCS*, pages 473–488. Springer-Verlag, October 2-5 2005.

[116] Kenneth J. Turner. Representing and analysing composed web services using CRESS. *Network and Computer Applications*, 30:541–562, 2007.

[117] H.M.A. van Beek. An algebraic approach to transactional processes. CS-Report 02/18, Department of Mathematics and Computing Science, Technische Universiteit Eindhoven, Dec 2002.

[118] Franck van Breugel. An introduction to metric semantics: operational and denotational models for programming and specification languages. *Theoretical Computer Science*, 258(1-2):1–98, May 2001.

[119] Rob J. van Glabbeek. Notes on the methodology of CCS and CSP. *Theoretical Computer Science*, 177(2):329–349, 1997.

[120] Mirko Viroli. Towards a formal foundation to orchestration languages. *Electronic Notes in Theoretical Computer Science*, 105:51–71, 2004.

[121] Glynn Winskel. *The formal semantics of programming languages: an introduction.* Foundation of Computing Science. MIT Press, 1993.

[122] Jim Woodcock and Ana Cavalcanti. The Semantics of Circus. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *Proceedings of 2nd International Z and B Conference (ZB'02)*, volume 2272 of *LNCS*, pages 184–203, France, 2002. Springer-Verlag.

[123] Yingzhou Zhang and Baowen Xu. A survey of semantic description frameworks for programming languages. *ACM SIGPLAN Notices*, 39(3):14–30, Mar 2004.

[124] Huibiao Zhu, Jonathan P. Bowen, and Jifeng He. Deriving operational semantics from denotational semantics for Verilog. In *8th Asia-Pacific Software Engineering Conference (APSEC 2001)*, pages 177 – 184. IEEE Computer Society, 4-7 Dec 2001.

[125] Huibiao Zhu, Jonathan P. Bowen, and Jifeng He. From operational semantics to denotational semantics for Verilog. In Tiziana Margaria and Thomas F. Melham, editors, *CHARME 2001*, volume 2144 of *LNCS*, pages 449–466, 2001.