Implementation of the stable revivals model in CSP-Prover.

Devadoss, Gift Samuel

Implementation of the Stable Revivals Model in CSP-Prover

D. Gift Samuel

A thesis submitted to the University of Wales in candidature for the degree of Master of Philosophy

Swansea University
Prifysgol Abertawe

Department of Computer Science
Swansea University

March 2008

To my father *S. Devadoss*

Courage, brother, do not stumble,
Though thy path be dark as night;
Theres a star to guide the humble:
Trust in God and do the right.
Let the road be rough and dreary,
And its end far out of sight,
Foot it bravely; strong or weary.
–*Norman Macleod*

Abstract

This thesis presents an implementation of the recently developed stable revivals model $\mathcal{R}$ in CSP-Prover. The stable revivals model is a new semantic model of the process algebra Communicating Sequential Processes (CSP). Bill Roscoe developed this new model of CSP in 2005 in order to capture process properties essential for Component Based Systems Design. On the practical side, the model $\mathcal{R}$ is developed to reason about responsiveness and stuck-freeness. These properties are vital for modular reasoning of Component Based Systems Design and other distributed systems.

CSP-Prover provides a deep encoding of the process algebra CSP in the generic theorem proving environment Isabelle. CSP-Prover can be used to prove refinements on infinite state systems and generic in the underlying architecture. Currently, it fully implements the stable failures model and the traces model of CSP. In this thesis, we extend CSP-Prover by implementing the stable revivals model to provide tool support for responsiveness and stuck-freeness. On the theoretical side, our implementation also yields a machine verification of the model $\mathcal{R}$'s soundness as well as of its expected properties.

We present a faithful and running implementation of the stable revivals model in the proof tool CSP-Prover. This requires certain changes with respect to Roscoe's original model in the implementation. We also present an improvement to Roscoe's model which allows to remove restrictions and implement the improved model in CSP-Prover.

Acknowledgements

First and foremost, I thank God for making this project a success and to finish. Next I thank my supervisor Dr. Markus Roggenbach for selecting me for this project and trusting me. I would like to thank him for introducing me to CSP. I would like to thank for his guidance, expertise, and encouragement. I would also like to thank Dr. Yoshinao Isobe for his constant help, insights, suggestions and guidance throughout this project. Without his help, I would not have completed my thesis by this time. I am grateful and thankful both Markus and Yoshinao for helping me to focus on this project. I would like to thank EPSRC for their finanicial support.

It is my pleasure to thank other group members in "Processes and Data" led by Markus Roggenbach. I thank Andy Gimblett for his help on anything at anytime. I thank Temesghen Kahsai for his useful discussion on progress of the whole project and helping in understand Isabelle. I thank Liam O'Reilly for his kindness and help in everything.

I thank University of Wales Swansea's Department of Computer Science for providing me all the facilities to finish my thesis well. In particular I would like to thank all the members in Hauptseminar seminars for giving me the opportunity to pariticipate. In particular, I thank Dr. Arnold Beckmann, Dr. Matt Jones, for their encouragement and advice throughout this project. I would also like to thank Dr. Christoph Lüth and Dr. Anton Setzer for correcting my thesis, and suggesting improvements. I would also like to thank them for providing valuable feedbacks to improve the thesis.

I would also like to thank other fellow research students in Department of Computer Science for this support throughout my M.Phil . Over the last few years, I have interacted with a lot of researchers. It has been a great inspiration for my work. In particular, I would like to thank Prof. R. Ramanujam, Prof. Supratik Chakraborty, and Dharmendra Kumar Yadav. I would also like to thank my house mates David Flak, Javier, Virgine, and Liu. I would like to thank to the bible house group meeting member, Vivian, Suzanne, Judith, Graham Young, Uki, Winfred, Juliet, etc. Other friends in Swansea Digesh, Sumit, Swati, Chaitali, Haris Rehman, etc for providing me relaxing environment in their house. I would also grateful to people in Swansea for their friendliness.

Finally, I would like to thank my parents and sister, D. Merlin, for their constant encouragement, love and support throughout this project from 5000 miles away. I must have to thank God for giving me the wonderful father and supportive family.

Table of Contents

List of Figures

Chapter 1

Introduction

Contents

1.1 Introduction

Verification of concurrent systems is an area of major research in computer science. The task of verification is to guarantee that an implementation of a system satisfies its specification. Concurrent systems consist of a set of processes that communicate with each other to perform some common task. The mode of execution and the mode of communication may differ from one system to another. [CGP99]. Concurrent systems are ubiquitous: from train controllers to avionics; from modern home appliances to automobiles; in business domain over enterprise supply management systems to flight management systems; from telecommunication systems to web services on the Internet; in critical systems like heart pacemakers to nuclear reactor controllers.

Formal methods [Ros98] analyse systems using mathematically rigorous techniques. In formal methods, specifications and implementations of systems are represented in notations that have clear semantics. It is then proved that the implementations meet their specifications using formal verification techniques. These kinds of techniques give assurance on the quality of systems.

Concurrent systems are complex. Formal verification of complex concurrent systems using mathematically rigorous techniques is a time consuming, tedious, and error-prone process. Hence computer-based tool support is useful for making the task easier. There are broadly two different ways to verify systems: model checking and theorem proving.

In model checking, an implementation of the system to be verified is usually modelled as finite state systems and specifications are formalised by writing temporal logic properties. The reachable states of the systems are traversed in order to verify the properties expressed in specifications [CGP99]. Model checking is an algorithmic method to verify if a given implementation of the system satisfies its specification. If the implementation satisfies its specification, it returns true. If the implementation does not satisfy its specification, it produces a counter example for the failure. Model checking is fully automatic; however, most of model checking tools can verify only systems that are finite states. Nowadays for model checking tools, models need not be finite states, and requirements can be specified in the variety of other languages. Moreover, model checking tools cannot verify very large systems due to the so-called state explosion problem that limits the application of model checking tools. FDR [Lim07], SPIN [Hol97], NuSMV [CCGR99], etc., are some examples of model checking tools. There have been many techniques to verify very large systems. Symbolic model checking is one of those techniques. In symbolic model checking, sets of states are represented by efficient data structures like Ordered Binary Decision Diagrams [Bry92]. It has been shown that symbolic model checking can be used in verifying industrial size systems. On the other hand, theorem proving can be used to verify infinite state systems. However, it can be automatic in restricted systems. In theorem proving, implementations and specifications are represented in some logic and then it is interactively proved that the implementations satisfy their specifications, hence theorem proving needs some skills. The latest advances in theorem prover make theorem proving easier and adds support for automatic proof generation. This thesis focuses on providing the tool support for the specifications of concurrent systems using theorem proving techniques.

Process algebra [Bae05, BPS01] is a formal method to specify and verify concurrent systems. Communicating Sequential Processes (CSP) [Hoa85, Hoa06, RBH81, RSG$^+$01, AJS05, Ros98], Calculus of Communicating Systems (CCS) [Mil89], π-calculus[3] [Mil99], the Algebra of Communicating Processes (ACP) [BK89] are well-known process algebras. CSP was designed by C.A.R. Hoare. A theory for CSP was later developed by C.A.R. Hoare, A. W. Roscoe and S.D. Brookes [RBH81]. CSP is a notation for describing systems of parallel agents that communicate by passing messages between them [RSG$^+$01, Ros98]. CCS was developed by R. Milner. A primary goal in the original design of CCS was to design and codify a minimal set of basic primitive agents and operators, which are capable in combination of describing all the characteristic phenomena encountered in the study of the interaction of concurrent agents [Hoa06]. CSP and CCS influenced one another throughout their development [Fid]. π-calculus is a process calculus developed by R. Milner and others as a continuation of the body of work on the process calculus CCS (Calculus of Communicating Systems). π-calculus was developed to describe concurrent computations whose configuration may change during the computation [Mil99]. ACP [BK89] is fundamentally an algebra to reasoning about concurrent systems. It was developed by J. Bergstra and J. W. Klop to describe systems in terms of algebraic approach. In this thesis, we focus on CSP.

Over the last decades, many process algebras have been developed to reason about security protocols, biological process, mobile processes, etc. Even though the different process calculi are designed for different purposes, all the process algebra languages share the following three major properties [Bae05]:

- Compositional modelling: Complex processes are built by combining basic primitive operators,
- Operational semantics: Meanings of processes are defined using Plotkin-styled operational semantics that describes the processes in terms of the single executions of steps.
- Behavioural reasoning via equivalences and preorders: Behavioural relation of processes can be related through process equivalence and processes refinement.

The meaning for process algebras can be defined in more than one way: algebraic semantics, denotational semantics or operational semantics. In algebraic semantics, which sometimes is also called axiomatic semantics, properties of operators are defined by axioms and laws. ACP follows algebraic semantics to define the meaning of the operators. Some of the basic laws in ACP are presented in the following where p_1, p_2 and p_3 are processes in ACP, and $+$ denotes the non-deterministic choice operator.

$$\begin{array}{rl} p_1 + p_2 = p_2 + p_1 & \textit{Commutativity} \\ (p_1 + p_2) + p_3 = p_1 + (p_2 + p_3) & \textit{Associativity} \\ p_1 + p_1 = p_1 & \textit{Idempotence} \end{array}$$

In operational semantics, processes are interpreted based on labelled transition systems. Equivalence and refinement are usually defined as bi-simulations and simulations. CCS follows the operational semantics approach to assign the meaning to the processes.

In denotational semantics, processes are interpreted by mapping the processes to mathematical objects. The collection of all mathematical objects in the denotational semantics is called the domain. Equivalence and refinements are defined by ordering relations on the domain. It is a general tenet of CSP to define semantics of processes using denotational semantics, but meaning can be assigned using operational semantics. In this thesis, we focus on embedding the denotational semantics of a new model of CSP in a higher order logic.

Mike Gordon et al [BGG+92] states the benefits that we get in embedding a language in higher order logic:
"The general technique of embedding a conventional notation, such as a hardware description language, in a mechanised formal systems, such as HOL, offers several possible benefits:

- formal definition of the semantics of various notation;
- mechanised support for syntax and type checking;
- a framework for establishing meta theorems about the notations (such as consistency);
- support for formal proof about programs;
- derivation of proof rules for notation (such as equational transformation);
- verification of compilers."

Thus embedding a process algebra in a theorem prover provides mechanical support for the definition of semantics. Proving theorems about the process algebra can help to verify correctness of the process algebra itself and to verify systems designed in the process algebra. [BGG+92] classifies the embedding of a language in a mechanised formal system in two ways:

(i) shallow embedding and

(ii) deep embedding.

In a deep embedding, the syntax and semantics of a language are embedded in a mechanised formal system. The mapping from the syntax of a language to the semantics is defined as a function in the formal system. In a deep encoding, the user can specify the syntax of a language as a new datatype and then define the semantics of the language. CSP-Prover has a deep encoding of the process algebra CSP. CSP-Prover is a theorem-proving tool to analyse CSP processes and CSP's properties. In this thesis, we embed a new semantic model of CSP in CSP-Prover. We will study deep embedding further when we discuss the architecture of CSP-Prover in Chapter 2.

In a shallow embedding, only the semantics of a language is defined in the formal system and the user provide a user-interface that parses the syntax of a language directly to the semantic structures. In a shallow embedding, we can only prove theorems in the embedded language [BG95].

Compared to a deep embedding, a shallow embedding saves cost as the syntactic component is not to be implemented. Also, in general it is assumed that in an shallow embedding theorems within the language are easier to prove. Experience with HOL-CSP (shallow encoding) and CSP-Prover (deep encoding) seems to indicate the opposite: Having the CSP-Syntax available, CSP-Prover allows for inductive proves along the syntactic structure. In the CSP context, this has shown to be a powerful and necessary proof principle. Furthermore, theorems on the language can only be achieved by a deep embedding.

The most notable difference between these two kinds of embedding is the interpretation of types in the language. In a shallow embedding, types in the embedding language are identified with the types of the formal system. Hence, the type correctness (well definedness of semantic) follows easily. In contrast, deep embedding identifies types in the language with types induced by the types in the formal system. In a deep embedding, this is quite difficult as the user has to prove well formedness and develop their own proof support, whereas in a shallow embedding, the semantics is defined in the logic of the theorem prover, and hence well formedness follows naturally. In a deep embedding, theorems about embedded languages are provable, but not in a shallow embedding.

1.1.1 Related Work

Mechanisation of process algebras in theorem provers is not a new work. Most of the major process algebras have been formalised in some theorem provers from the early nineties of the last century. In this section, we discuss some of them. A subset of the process algebra CCS is formalised by Nesi [Nes92] in Higher Order Logic (HOL) [NPW06]. The formal theories for observational congruence and for a slight extension of Hennessy-Milner modal logic over pures CCS (with no value passing) are embedded in the HOL logic to support verification and reasoning about processes specifications.

Camilleri [Cam90] mechanised the traces model of CSP in HOL to reason about CSP processes. Mechanisation follows a definitional approach to avoid inconsistency laws, hence the logic is extended conservatively. Later, he mechanised a variation of the failures-divergence model of CSP in HOL [Cam91]. In both implementations, the sequential operator of CSP is not included. This is one of the early mechanisation of a concurrent specification language in higher order logic.

In [TW97], Tej and Wolff formalised CSP in Isabelle/HOL [NPW02]. It is a shallow embedding of the failures divergence model based on the CPO approach. Theories on CPO, continuity, Knaster-Tarski and fixed-point induction are formalised to reason about recursive CSP processes. The embedding included the sequential operator of CSP. The embedding revealed an error in the type correctness of the sequential operator. Like CSP-Prover, it allows to reason about infinite communication alphabets and unbounded non-determinism. A corrected model has been formally proven consistent with Isabelle/HOL.

Dutertre and Schneider [DS97] formalised the traces model of CSP in PVS [ORS92] to reason about authentication protocols. Recently, Wei and Heather [WH05] extended the formalisation to reason about the stable failures model of CSP in PVS. They have proved determinism and deadlock freedom of the asymmetric dinning philosophers problem with an arbitrary number of philosophers and have proved an example of an industrial-scale application 'virtual network' with any number of dimensions to be deadlock-free.

Very recently, Kammuller [Kam07] formalised the failures divergence model of CSP in Isabelle/HOL. Embedding uses many standard features of theorem prover like Tarski's fixed-point theorem provided in Isabelle/HOL. It has been developed to keep the embedding lightweight and as simple as possible for teaching purpose and easy for modification.

1.2 Motivation of the project

Communicating Sequential Processes (CSP) [Hoa85, Ros98, RBH81, RSG$^+$01, AJS05] is one of the process algebras, which has widely been used to describe concurrent systems. By fixing one syntax and varying behaviours observed on the processes, we can get different model [RRS06, Ros98].

Mathematical models of CSP processes are constructed based on properties and behaviours that we are interested in. The traces model ($\mathcal{T}$), the failure divergences model ($\mathcal{N}$), and the stable failures model ($\mathcal{F}$) are some examples of well-known and well-studied mathematical models in CSP. The traces model suits well to reason about safety properties. The failure-divergences and the stable failures model are developed mainly to reason about liveness properties in CSP. Recently, Roscoe [RRS06, Ros07] developed a new model called the stable revivals model $\mathcal{R}$ to reason about responsiveness and stuck-freeness in distributed systems. Responsiveness and stuck-freeness are properties of modular reasoning or compositional reasoning in Component Based Systems. The aim of this project is to implement the newly developed model $\mathcal{R}$ in CSP-Prover [IR07b, IR05].

CSP-Prover provides a deep encoding of the process algebra CSP in the generic theorem proving environment Isabelle. CSP-Prover can be used to prove refinement on infinite state systems. Currently, it implements the stable failures model and the traces model in CSP. In this thesis, we extend CSP-Prover by implementing the stable revivals model to provide tool support for this model. Errors found in the semantic function of the sequential operator in the model $\mathcal{N}$ [TW97] and algebraic laws in model $\mathcal{F}$ [IR05] show that the mechanisation of CSP models can reveal errors in well-established theories. It shows that definition of models will be 'complete' [Ros06], once they have been defined in a mechanised theorem prover. To build a sound logical system, the semantics of the stable revivals model is implemented by a definitional approach similar to other models in CSP-Prover. The benefits of the project are as follows:

- On the theoretical side, for the CSP community, embedding allows one to verify mechanically the important properties in the stable revivals model. One of the major properties is mechanical verification of the definition of semantics. Especially, the type correctness of the semantic clauses, continuity of the semantic functions, validation of algebraic laws.
- On the practical side, for the software engineering community, implementing the stable revivals model in CSP-Prover allows one to reason about responsiveness and stuck-freeness properties in a theorem-proving environment. Thus, this project delivers tool support for reasoning about responsiveness. Responsiveness says that in an interaction between two processes, one process will not cause another one to deadlock by not responding to it when expected. Stuck-freeness of a pair of interacting processes means the combination terminates successfully without leaving one process hanging. Responsiveness has been identified as one of the key non-functional properties of large-scale systems in the UK grand challenge problem [UKG08].

- It is shown in [Ros07] that some important algebraic laws like $\sqcap$-$\Box$-distributivity fail in the model. By embedding, we can mechanically prove algebraic laws in CSP in the CSP-Prover in a sound way. Manual verification of these algebraic laws is an error-prone, tedious, and complex task.
- Our work extends CSP-Prover tool with yet another CSP model. Thus it shows that CSP-Prover is a versatile tool and easily extendable.

1.3 Thesis Outline

In this thesis, we present an implementation of the stable revivals model in CSP-Prover. The implementation of the model has the following steps in CSP-Prover proof infrastructure:

1. Creating a new type to represent the domain of the model.
2. Proving that the domain of model is a complete partial order.
3. Encoding the semantic function of the model.
4. Proving the type correctness of the semantic functions.
5. Proving that the semantic function are continuous.
6. Providing a proof infrastructure for recursive process.
7. Proving the basic algebraic laws (validating step laws).

The implementation of the first three steps is given in Chapter 5 and the implementation of the next three steps is given in Chapter 6. The validation of the basic algebraic laws is given in Chapter 7. First, we present the important results of this thesis and then outline the organisation of this thesis.

1.3.1 Mistakes Found

In Chapter 5, we will see that the definition of the domain of the stable revivals model needs to be added with a condition given in [RRS06]. This condition is needed to prove the type correction of the hiding operator in Chapter 6.

The implementation has revealed the following mistakes:

- The step laws of `STOP` and the external choice operator fail. Counter examples for the failure are given in Chapter 7. These mistakes are corrected by including a semantic clause in the definition of deadlock of the prefix choice operator.
- The step of the renaming operator fails with the modified semantics of the prefix choice operator. A counter example[1] for the failure is given in Chapter 7. This mistakes is corrected by modifying the semantic clause of deadlock of the renaming operator.
- The type correctness of the renaming operator fails when the renaming relation is infinite. A counter is given in Chapter 6.

[1]This is found by Dr. Anton Setzer while discussing this problem.

- An improved stable revivals model is given in 8. The stable revivals model [Ros07] is given assuming Σ is finite. In the improved stable revivals, Σ can be infinite. We proved the type correctness and continuity for the both models.

1.3.2 Thesis organisation

This thesis is organised as follows.

Chapter 2 introduces CSP and CSP-Prover, explaining the syntax that we use in the thesis. We will discuss the models of CSP, which come closest to the stable revivals model. In this chapter, we introduce the traces model and the stable failures model. We motivate each model using examples.

In Chapter 3, we study the motivation of the stable revivals model with some examples. In this chapter, we present the formal definition of stuckfreeness and responsiveness. In Chapter 4, we present the semantics of the stable revivals model and explain some of its properties.

Chapter 5 describes how we implement the stable revivals model in CSP-Prover. In this chapter, we explain the code of the implementation. In Chapter 6, we discuss the various properties proved in the implementation. We also discuss continuity and the type correctness of semantic functions. We give the proofs in detail and explain how they are implemented in CSP-Prover.

Chapter 7 focuses on various algebraic laws proved in this model. We present selected basic laws and selected step laws and prove to be correct them w.r.t. the stable revivals model using our implementation. We also explain the step laws in detail.

Chapter 8 gives examples for running tool in the stable revivals models. Finally, in Chapter 9, we summarise our work, and conclude the thesis with future work.

Chapter 2

Background

Contents

CSP is a language to describe concurrent systems. Processes in concurrent systems communicate with each other by engaging in events from an alphabet set Σ. The events occur instantaneously and are considered to be atomic. Processes communicate with other processes by means of synchronous or handshake communication where all the participating processes must agree on an event to happen. In this chapter, we explain about CSP and CSP-Prover.

This chapter is organised as follows. Firstly, we explain CSP and its syntax. Secondly, we discuss some semantic models of CSP. Thirdly, we focus on Isabelle theorem prover. Finally, we explain how CSP-Prover is implemented in Isabelle.

2.1 Syntax of CSP

Many dialects of CSP syntax are available in the literature. In this thesis, we use the syntax of CSP$_{\text{TP}}$ [IR07a] implemented in CSP-Prover, the subscript TP stands for Theorem Proving. In this section, we discuss the grammar of CSP$_{\text{TP}}$ and explain its dissimilarity with the core CSP discussed in [Ros07]. We give an intuitive meaning of each operators. Appendix A.1 shows the syntax of the core CSP language as given in [Ros07].

In CSP$_{\text{TP}}$, alphabets of communication events Σ can be arbitrary. The syntax for CSP$_{\text{TP}}$ is shown in Figure 2.1. CSP$_{\text{TP}}$ does not include the generalised internal non-deterministic choice operator $\bigsqcap$, instead it has replicated internal choice `! ! c . C • P(c)`.

Given an alphabet of communications Σ and the data type of natural numbers Nat, we form the set $Choice(\Sigma) = \mathbb{P}(\mathbb{P}(\Sigma)) \uplus \mathbb{P}(Nat)$, where $\uplus$ is a disjoint union of two sets. The replicated internal choice takes an index set $C \in Choice(\Sigma)$ as its parameter, thus $C \subseteq \mathbb{P}(\Sigma)$ or $C \subseteq Nat$. The set of processes is denoted by $Proc_{(\Pi,\Sigma)}$. The replicated internal choice is restricted to run over an indexed set of processes

$$P(.) : \mathbb{P}(\Sigma) \uplus Nat \Rightarrow Proc_{(\Pi,\Sigma)}$$

Another difference is that CSP$_{TP}$ includes the restriction operator ($\lfloor$) in the syntax.

$P ::=$	SKIP	%% successful terminating process
$\mid$	STOP	%% deadlock process
$\mid$	DIV	%% divergence
$\mid$	$a \rightarrow P$	%% action prefix
$\mid$	$?\,x : X \rightarrow P(x)$	%% prefix choice
$\mid$	$P \,\Box\, P$	%% external choice
$\mid$	$P \sqcap P$	%% internal choice
$\mid$	$!!\, c : C \bullet P(c)$	%% replicated internal choice
$\mid$	IF b THEN P ELSE P	%% conditional
$\mid$	$P \setminus X$	%% hiding
$\mid$	$P[[R]]$	%% relational renaming
$\mid$	$P \fatsemi P$	%% sequential composition
$\mid$	$P \bigtriangleup P$	%% interrupt
$\mid$	$P \rhd P$	%% timeout
$\mid$	$P \lfloor n$	%% depth restriction
$\mid$	$P \,\|[\, X \,]\|\, P$	%% generalized parallel
$\mid$	$\$p$	%% process name

where $X \subseteq \Sigma$, $C \in Choice(\Sigma)$, $b \in Bool$, $a \in \Sigma$, $R \in \mathbb{P}(\Sigma \times \Sigma)$, $n \in Nat$, and $p \in \Pi$(set of process names).

Figure 2.1: Syntax of basic CSP$_{TP}$ processes in CSP-Prover.

CSP processes are composed from basic primitive processes, atomic events Σ, and the natural numbers $\mathcal{N}$. The primitive processes in the language are STOP, DIV, and SKIP. SKIP is a process that successfully terminates and does not perform any events from Σ. STOP is a process which does nothing and represents deadlock. Deadlock is the state of a process where no further action is possible. Thus, STOP turns out to be a model for deadlock. It might look useless, but is very useful in writing specifications. DIV is a process that engages in an infinite sequence of invisible actions. It is similar to STOP, however it performs internal actions that are not visible to its environment.

Given an event a and a process P, then $a \rightarrow P$ is a prefixing process that engages in the event a and then behaves like the process P. The only way a process can communicate or interact with the environment is by communicating events.

If $X \subseteq \Sigma$ is a set of events, and $P(x)$ is a process for each event x in the set X, then $?\ x : X \rightarrow P(x)$ is a prefix choice process that is initially able to perform an event a from X and after engaging in a, it behaves as $P(a)$. The prefixing process $a \rightarrow P$ can be represented as $?\ x : \{a\} \rightarrow P(a)$. The primitive process STOP can be defined in terms of $?\ x : \emptyset \rightarrow P(x)$. This is also called the step law for STOP.

There are two forms of binary choice available in CSP$_{TP}$. Given two processes P and Q. The external choice process $P \,\Box\, Q$ gives a choice to the environment to execute processes P or Q based on the initial events of P and Q. The behaviour of the process $(a \rightarrow P) \,\Box\, (b \rightarrow Q)$ depends on the initial events from the environment. If a and b are same, then choice is resolved in a non-deterministic way. Another choice operator in CSP$_{\text{TP}}$ is the internal choice operator.

The behaviour of the internal choice process $P \sqcap Q$ does not depend on the environment. One of the processes is selected in a non-deterministic way. It is not possible to tell which choice will be made. No fairness condition is assumed. Given an index set $C \in \mathit{Choice}(\Sigma)$ such that a process $P(c)$ is defined for each element $c \in C$, then the replicated internal choice process `!!c.C • P(c)` selects one of processes $P(c)$ in a non-deterministic way and executes $P(c)$.

If b is a boolean expression, then IF b THEN P ELSE Q is a conditional process. If b evaluates to True, then P will be performed, else Q will be performed.

If P is a process and X is a set of events, then $P \setminus X$ is a hiding process that makes the events X in process P unobservable to the environment. The process $P \setminus X$ performs all the external events as P does, except that the events in X are performed as internal events. Hiding is useful in abstraction and prevents the environment from engaging in or observing. It is one source of non-determinism. In some process algebras like CCS, hiding is combined with other operators.

Another useful operator is the renaming operator. If $R \subseteq \Sigma \times \Sigma$ is a relation over the alphabet and P is a process, $P \,[\![R]\!]$ is a renaming process formed by renaming x to y in P for all $(x, y) \in R$. If an event a occurring in P is not in the domain of R, then renaming behaves like the hiding operator.

If more than one event is mapped into a single event, then non-determinism is introduced. Given a process P and a natural number n, the restriction operator ($P \lfloor n$) is a process that performs like P for its first n events and then stops.

Given two processes P and Q, then $P \fatsemi Q$ is a sequential process that behaves as P until P terminates successfully and after that the process $P \fatsemi Q$ behaves like Q. The interrupt operator ($\triangle$) is similar to the sequential operator, the combined process $P \triangle Q$ behaves like P until the environment interact one of its initial events of Q, and then $P \triangle Q$ stops performing like P and starts behaving like Q. The timeout operator ($P \rhd Q$) offers P for a short time, and if none of the initial events of P are offered it opts to behave like Q.

All the operators that we have so far seen allow describing sequential processes in a network. Allowing two or more processes to interact with each other makes concurrent processes and also makes our analysis complex and interesting. In an interaction between processes P and Q, we usually want only some events to interact or synchronise, hence CSP gives freedom to specify a synchronised set explicitly. The generalised parallel process $P \,[\![X]\!]\, Q$ describes that all the events in the set $X \subseteq \Sigma$ must be synchronised and events outside X can proceed independently.

CSP has three more parallel operators derived from the generalised parallel operator. These operators differ on the synchronisation of events; the parallel operators varies from processes that do not synchronise at all to processes that synchronise all the events. The operator that does not synchronise any event is called the interleaving operator $P \,|||\, Q$. It can be defined as syntactic sugar of the generalised parallel operator

$$P \,|||\, Q \equiv P \,[\![\emptyset]\!]\, Q.$$

In the interleaving operator $P \,|||\, Q$ where no synchronisation is required, both the processes P and Q perform all events completely independent of each other.

On other extreme, if all the events from Σ are synchronised then operator is called synchronised parallel operator $\|$. In the synchronised parallel process $P \parallel Q$, for an event to happen, both the processes P and Q must synchronise on it. It can be defined as

$$P \parallel Q \equiv P \,[\![\Sigma]\!]\, Q.$$

Between the synchronised parallel operator and the interleave operator is the alphabetised parallel operator. In the alphabetised parallel process $P \,[\![X, Y]\!]\, Q$, the process P can communicate events in X and the process Q can communicate events in Y, and both are synchronised on $X \cap Y$. It can be created by the generalised parallel operator by the following equation

$$P \,[\![X \mid Y]\!]\, Q \equiv (P \,[\![\Sigma \setminus X]\!]\, SKIP) \,[\![X \cap Y]\!]\, (Q \,[\![\Sigma \setminus Y]\!]\, SKIP).$$

In the rest of the thesis, we focus only on the generalised parallel operator.

Up to now, process can describe only the finite behaviours. In order to deal also with infinite behaviours, we introduce now the concept of recursion using process names. In the syntax of CSP$_{\text{TP}}$, process names are introduced. Process names are defined in the left hand side and used in the right hand side. A process is defined as an equation of the form

$$p(x_1, x_2 \ldots x_k) = P$$

where $p(x_1, x_2 \ldots x_k)$ is a process name, $x_1, x_2, \ldots x_k$ are global variables and P is a process which may include the process name p as $\$p$. It enables us to define recursive processes which is useful for describing complex processes.

Consider for example the recursive process

$$As = a \rightarrow \$As \,\Box\, b \rightarrow \mathtt{SKIP}$$

$\$As$ is a process which performs any number of a and then terminates with b.

There are two kinds of variables in CSP$_{\text{TP}}$:
(i) Local variables, and
(ii) Global variables

Global variables are declared in the left hand side along with process names. Its scope is the whole right hand size of the equation. Consider for example the process

$$Count(n : \mathcal{N}) = (n \rightarrow P) \operatorname{\S} Q(n)$$

In the above example, n is a global variable and its scope is the whole process $(n \rightarrow P) \operatorname{\S} Q(n)$. *Count* is called a parameterized process. For local variables, CSP$_{\text{TP}}$ follows a declarative semantics like CSP described in [Ros98]: an identifier gets its value at the point where it is declared and keeps the same value throughout that scope. The scope of local variables x in the process $x \rightarrow P;\ Q$ is within P only. In the process $?n : A \rightarrow P;\ Q$, the process creates a new identifier n and its scope is valid within P. Each variable will be substituted by a concrete value by the time it performs the action.

We give the meaning of the process names while explaining the formal semantics of the CSP processes. The well-formed terms in the language without a process name are called the *closed terms of* CSP or the *basic terms of* CSP . The processes in the language which may also have process names are called *terms of* CSP or CSP terms. *CSP* denotes the set of terms of CSP as defined in [Ros98].

2.2 Semantics of CSP

CSP has well defined denotational semantics and operational semantics. For some CSP semantic models, well-defined axiomatic semantics are available. A complete axiomatic semantics for the model $\mathcal{N}$ with bounded non-determinism over a finite alphabet is given in [Ros98]. In [IR05], a complete axiomatic semantics for the stable failures model with unbounded non-determinism over an alphabet of arbitrary size is given. The semantics of CSP is based on observation made on CSP processes. Observations of a process are usually events performed and not performed by the process. In some sophisticated models, internal events are also considered. This helps us to define and understand concepts like non determinism, livelock, etc., precisely. Each model has its own degree of complexity and expressiveness. In the following, we discuss the traces model and the stable failures model of CSP as these are closed related with the stable revivals model.

2.2.1 The Traces Model

The traces model is the simplest model of CSP and suites well for analysing safety properties. In the traces models $\mathcal{T}$, we record finite sequences of events that can be performed by a process at any arbitrary time.

In the traces model, each process is identified by a set $T \subseteq \Sigma^{*\checkmark}$ that satisfies the following conditions, where $\Sigma^{*\checkmark} = \Sigma^{*} \cup \{s ^\frown \langle\checkmark\rangle \mid s \in \Sigma^{*}\}$.

T1. T is nonempty;

T2. T is prefix-closed; i.e., $s ^\frown t \in T$, then $s \in T$.

Thus, all the processes can perform the empty trace $\langle\rangle$. A trace is an element of $\Sigma^{*\checkmark}$. First we introduce the domain of the traces model.

Definition 2.1: The domain of the traces model. Given an alphabet set Σ, the domain of the traces model $dom(\mathcal{T})$ is defined to be the set of all $T \subseteq \Sigma^{*\checkmark}$ satisfying the healthiness conditions $T1$ and $T2$.

The denotational semantics of the traces model is given by a function $traces_M$. The function $traces_M$ maps process CSP terms into subsets of $\Sigma^{*\checkmark}$. The type of the semantic function $traces_M$ is given by

$$CSP \longrightarrow Environment \longrightarrow dom(\mathcal{T})$$

The environment M is a function from processes names into the domain of the traces model. This helps us to define the semantics of recursive processes.

The function $traces_M$ is defined inductively over the grammar which means the meaning of process is calculated by its sub processes. We define the following notations that will be used in semantic function.

- $t_1 \parallel [X] \parallel t_2$ is inductively defined by:

$$\begin{array}{rl}
\langle x\rangle ^\frown t_1 \parallel[X]\parallel \langle x\rangle ^\frown t_2 & = \{\langle x\rangle ^\frown u \mid u \in t_1 \parallel[X]\parallel t_2\}\\
\langle x\rangle ^\frown t_1 \parallel[X]\parallel \langle x'\rangle ^\frown t_2 & = \emptyset\\
\langle x\rangle ^\frown t_1 \parallel[X]\parallel \langle\rangle & = \emptyset\\
\langle\rangle \parallel[X]\parallel \langle x\rangle ^\frown t_2 & = \emptyset\\
\langle\rangle \parallel[X]\parallel \langle\rangle & = \{\langle\rangle\}\\
\langle y\rangle ^\frown t_1 \parallel[X]\parallel \langle x\rangle ^\frown t_2 & = \{\langle y\rangle ^\frown u \mid u \in t_1 \parallel[X]\parallel \langle x\rangle ^\frown t_2\}\\
\langle y\rangle ^\frown t_1 \parallel[X]\parallel \langle\rangle & = \{\langle y\rangle ^\frown u \mid u \in t_1 \parallel[X]\parallel \langle\rangle\}\\
\langle x\rangle ^\frown t_1 \parallel[X]\parallel \langle y\rangle ^\frown t_2 & = \{\langle y\rangle ^\frown u \mid u \in \langle x\rangle ^\frown t_1 \parallel[X]\parallel t_2\}\\
\langle\rangle \parallel[X]\parallel \langle y\rangle ^\frown t_2 & = \{\langle y\rangle ^\frown u \mid u \in \langle\rangle \parallel[X]\parallel t_2\}\\
\langle y\rangle ^\frown t_1 \parallel[X]\parallel \langle y'\rangle ^\frown t_2 & = \{\langle y\rangle ^\frown u \mid u \in t_1 \parallel[X]\parallel \langle y'\rangle ^\frown t_2\}\\
& \quad \cup\{\langle y'\rangle ^\frown u \mid u \in \langle y\rangle ^\frown t_1 \parallel[X]\parallel t_2\}
\end{array}$$

  where t_1, t_2, u are traces, $X \subseteq \Sigma$, $x, x' \in X \cup \{\checkmark\}$, $y, y' \notin X \cup \{\checkmark\}$, and $x \neq x'$,

- we lift a relation R to traces over $\Sigma^{*\checkmark}$ by defining relation $[[R]]^*$ as follows:

$$\begin{array}{rl}
(s,t) \in [[R]]^* \Leftrightarrow & s = t = \langle\rangle \ \vee\\
& s = t = \langle\checkmark\rangle \ \vee\\
& s = \langle a\rangle ^\frown s \wedge t = \langle b\rangle ^\frown s \wedge (a,b) \in R \wedge (s,t) \in [[R]]^*
\end{array}$$

  where $R \subseteq \Sigma \times \Sigma$, s, t are traces, $a, b \in \Sigma$

- $(t \setminus X)$ is inductively defined by:

$$\begin{array}{rll}
\langle\rangle \setminus X & = \langle\rangle & \\
(\langle x\rangle ^\frown t) \setminus X & = t \setminus X & (\text{if } x \in X)\\
(\langle y\rangle ^\frown t) \setminus X & = \langle y\rangle ^\frown (t \setminus X) & (\text{if } y \notin X)
\end{array}$$

  where t is a trace, $X \subseteq \Sigma$,

- $[[R]]^{-1}(X)$ is defined as:

$$[[R]]^{-1}(X) = \{a \mid \exists\, b \in X.\ (a,b) \in R \vee a = b = \checkmark\}$$

  where $R \subseteq (\Sigma \times \Sigma)$ and X is a set of events.

- Restriction functions $T \downarrow n$, where n is a natural number, are defined as follows:

$$T \downarrow n = \{t \in T \mid |t| \leq n\}$$

  where t is a trace, $|t|$ returns the length of the trace t and $n \in \mathcal{N}$

The inductive definition of $traces_M$ on the process terms P is given by Figure 2.2.

SKIP terminates by producing the special event $\checkmark$ where $\checkmark \notin \Sigma$. The traces of $traces_M(\mathtt{SKIP})$ is either the empty trace $\langle\rangle$ or the trace which performs the termination event $\langle\checkmark\rangle$ successful. Hence, it has only two traces; one is the empty trace $\langle\rangle$ which denotes the trace before the engaging in the event $\checkmark$ in addition, the other is $\langle\checkmark\rangle$ the trace after the engaging in the event $\checkmark$. The traces of $traces_M(\mathtt{STOP})$ does not perform any event. Hence it has only one trace, the empty trace $\langle\rangle$. The traces

$$\begin{aligned}
\mathit{traces}_M(\mathtt{SKIP}) &= \{\langle\rangle, \langle\checkmark\rangle\}\\
\mathit{traces}_M(\mathtt{STOP}) &= \{\langle\rangle\}\\
\mathit{traces}_M(\mathtt{DIV}) &= \{\langle\rangle\}\\
\mathit{traces}_M(a \longrightarrow P) &= \{\langle\rangle\} \cup \{\langle a\rangle ^\frown t' \mid t' \in \mathit{traces}_M(P)\}\\
\mathit{traces}_M(?\, x : A \longrightarrow P(x)) &= \{\langle\rangle\} \cup \{\langle x\rangle ^\frown t' \mid t' \in \mathit{traces}_M(P(x)), x \in A\}\\
\mathit{traces}_M(P \,\Box\, Q) &= \mathit{traces}_M(P) \cup \mathit{traces}_M(Q)\\
\mathit{traces}_M(P \sqcap Q) &= \mathit{traces}_M(P) \cup \mathit{traces}_M(Q)\\
\mathit{traces}_M(!!\ c : C \bullet P(c)) &= \bigcup\{\mathit{traces}_M(P(c)) \mid c \in C\} \cup \{\langle\rangle\}\\
\mathit{traces}_M(\mathtt{IF}\ c\ \mathtt{THEN}\ P\ \mathtt{ELSE}\ Q) &= \text{if } c \text{ evaluates to } \mathit{True} \text{ then } \mathit{traces}_M(P) \text{ else } \mathit{traces}_M(Q)\\
\mathit{traces}_M(P \,\|[X]\|\, Q) &= \{t_1 \,\|[X]\|\, t_2 \mid t_1 \in \mathit{traces}_M(P),\ t_2 \in \mathit{traces}_M(Q)\}\\
\mathit{traces}_M(P \setminus X) &= \{t \setminus X \mid t \in \mathit{traces}_M(P)\}\\
\mathit{traces}_M(P[[R]]) &= \{t \mid \exists\, t' \in \mathit{traces}_M(P).\ (t', t) \in [[R]]^*\}\\
\mathit{traces}_M(P \operatorname{⨾} Q) &= (\mathit{traces}_M(P) \cap \Sigma^*)\\
&\quad \cup \{t_1 ^\frown t_2 \mid t_1 ^\frown \langle\checkmark\rangle \in \mathit{traces}_M(P),\ t_2 \in \mathit{traces}_M(Q)\}\\
\mathit{traces}_M(P \bigtriangleup Q) &= \mathit{traces}_M(P) \cup \{s ^\frown t \mid s \in \mathit{traces}_M(P) \cap \Sigma^*, t \in \mathit{traces}_M(Q)\}\\
\mathit{traces}_M(P \rhd Q) &= \mathit{traces}_M(P) \cup \mathit{traces}_M(Q)\\
\mathit{traces}_M(P \lfloor n) &= \mathit{traces}_M(P) \downarrow n\\
\mathit{traces}_M(\$p) &= M(p)
\end{aligned}$$

where $X \subseteq \Sigma$, $C \in \mathit{Choice}(\Sigma)$, $c \in \mathit{Bool}$, $a \in \Sigma$, $R \in \mathbb{P}(\Sigma \times \Sigma)$, $n \in \mathit{Nat}$, and $p \in \Pi$(set of process names). M is an environment.

Figure 2.2: Semantic clauses for the model $\mathcal{T}$ in CSP$_{\text{TP}}$.

of $\mathit{traces}_M(\mathtt{DIV})$ is same as the traces of $\mathit{traces}_M(\mathtt{STOP})$ as it does not perform any event which is externally observable. DIV engages in an internal event continuously.

The traces of $\mathit{traces}_M(\mathtt{a} \longrightarrow \mathtt{P})$ are either the empty trace $\langle\rangle$ or traces which perform the event a followed the traces of P. A trace of $\mathit{traces}_M(?x : A \longrightarrow P)$ is either a trace which does perform any event or perform the event $a \in A$ followed by a trace of $P[a/x]$. $P[a/x]$ represents the substitution of the value a for all free occurrence of the local variable x. The scope of the variable x is until P terminates.

The traces of $\mathit{traces}_M(P \,\Box\, Q)$ are either the traces of P or the traces of Q. The traces of $\mathit{traces}_M(P \sqcap Q)$ and the traces of $\mathit{traces}_M(P \rhd Q)$ are calculated similar to $\mathit{traces}_M(P \,\Box\, Q)$. Hence, the traces model does not distinguish between the internal choice and the external choice operator. The process

$$?\ x : A \cup B \longrightarrow P(x)$$

has the same behaviour as the process

$$?\ x : A \longrightarrow P(x)\ \Box ?\ x : B \longrightarrow P(x).$$

The traces of the generalised parallel process $P \,\|[\, X \,]\|\, Q$ are given by the set $\{t_1 \,\|[\, X \,]\|\, t_2 \mid t_1 \in \mathit{traces}_M(P) \wedge t_2 \in \mathit{traces}_M(Q)\}$. All the traces in $P \,\|[\, X \,]\|\, Q$ are a combination of traces of P and Q such that events in X are shared and the rest can happen independently. We look at some examples for it.

$$\mathit{traces}_M(a \longrightarrow b \longrightarrow \mathit{Skip} \,\|[\,\{a\}\,]\|\, a \longrightarrow c \longrightarrow \mathit{Skip}) =$$

$$\{\langle\rangle, \langle a\rangle, \langle a, b\rangle, \langle a, b, c\rangle, \langle a, c\rangle, \langle a, c, b\rangle, \langle a, b, c, \checkmark\rangle, \langle a, c, b, \checkmark\rangle\}$$

where

$$traces_M(a \rightarrow b \rightarrow Skip) = \{\langle\rangle, \langle a\rangle, \langle a, b\rangle, \langle a, b, \checkmark\rangle\},$$

$$traces_M(a \rightarrow c \rightarrow Skip) = \{\langle\rangle, \langle a\rangle, \langle a, c\rangle, \langle a, c, \checkmark\rangle\}.$$

In the example, the event a in the trace $\langle a, b, c, \checkmark\rangle$ comes from two traces $\langle a, b, \checkmark\rangle$ and $\langle a, c, \checkmark\rangle$. $\checkmark$ is the special event on which all CSP parallel operators effectively synchronise and it denotes distributed termination. For example, the process

$$a \rightarrow SKIP \mathbin{|[\emptyset]|} b \rightarrow SKIP$$

has the same behaviour as

$$a \rightarrow b \rightarrow SKIP \mathbin{\Box} b \rightarrow a \rightarrow SKIP.$$

Consider for example in the alphabetised parallel operator, the process

$$a \rightarrow c \rightarrow SKIP \mathbin{|[\{a, c\} \mid \{b, c\}]|} c \rightarrow b \rightarrow SKIP$$

has the same behaviour as the process $a \rightarrow c \rightarrow b \rightarrow SKIP$. Initially, the event a from the left hand size is performed and then both communicating processes synchronise on the event c. Then, it performs the event b from the right side.

Consider for example, the process

$$a \rightarrow b \rightarrow SKIP \mathbin{|[\{a, b\}]|} a \rightarrow b \rightarrow SKIP.$$

Above process has the same behaviour as $a \rightarrow b \rightarrow SKIP$. Consider another example where processes do not synchronise, the process $a \rightarrow SKIP \mathbin{|[\{a, b\}]|} b \rightarrow SKIP$ has the same behaviour as $STOP$. This is because the events a and b do not synchronise.

The traces of $traces_M(P \setminus X)$ is a set of all traces of P such that the events in X are removed from the traces sequences. It is best illustrated with an example:

$$traces_M(a \rightarrow b \rightarrow SKIP \setminus \{a\}) = \{\langle\rangle, \langle b\rangle, \langle b, \checkmark\rangle\}$$

where $traces_M(a \rightarrow b \rightarrow SKIP) = \{\langle\rangle, \langle a\rangle, \langle a, b\rangle, \langle a, b, \checkmark\rangle\}$.

In more elaborate models than the traces model, the hiding can be the source for internal non-determinism. In the process

$$((a \rightarrow P) \mathbin{\Box} (b \rightarrow Q)) \setminus \{a, b\} = P \setminus \{a, b\} \sqcap Q \setminus \{a, b\}$$

non-determinism is introduced because we do not how whether the events a or b has happened.

The traces of $traces_M(P[[R]])$ is set of all traces of P such that the event x is replaced by the event y for $(x, y) \in R$. The number of traces in $(P[[R]])$ may be more than the number of traces in P. Consider for example,

$$traces_M(a \rightarrow STOP[[\{(a, b), (a, c)\}]]) = \{\langle\rangle, \langle b\rangle, \langle c\rangle\}.$$

In the examples, a is mapped to b and c, hence we get two traces $\langle b\rangle$ and $\langle c\rangle$ from the single trace $\langle a\rangle$. Consider for example the process

$$a \rightarrow SKIP \mathbin{|[\{(a, b), (a, c)\}]|} = b \rightarrow SKIP \mathbin{\Box} c \rightarrow SKIP.$$

The traces of the sequential process $P \mathbin{;} Q$ is the set of all traces of P in which $\checkmark$ does not appear and the set of all traces of P concatenated with the traces of Q such that $\checkmark$ is removed from traces of P. Hence, $\checkmark$ from the process Q only appear at the end for terminating Q process. In the sequential operator, $\checkmark$ from the first process gets hidden and it will look as if $\checkmark$ has not happened. $P \mathbin{;} Q$ terminates only if Q terminates successfully. Consider for example the process

$$traces_M(a \longrightarrow Skip \mathbin{;} b \longrightarrow Skip) = \{\langle\rangle, \langle a\rangle, \langle a, b\rangle, \langle a, b, \checkmark\rangle\}$$

where $\checkmark$ from $a \longrightarrow Skip$ is hidden. The traces of the interrupt process $P \mathbin{;} Q$ are the set of all traces of P, and the set of all traces of P concatenated with the traces of Q such that the traces of P should not contain $\checkmark$. Unlike the sequential operator, in the interrupt operator $P \triangle Q$ the complete traces of P is included as it is also necessary to prevent the transfer once the process P has terminated. Consider the previous combined with the interrupt operator.

$$traces_M(a \longrightarrow Skip \triangle b \longrightarrow Skip) = \{\langle\rangle, \langle a\rangle, \langle a, \checkmark\rangle, \langle b\rangle, \langle b, \checkmark\rangle, \langle a, b\rangle, \langle a, b, \checkmark\rangle\}$$

In this case, $\checkmark$ from $a \longrightarrow Skip$ is included in the interrupt operator.

The traces in the depth restriction process $P \lfloor n$ is a set of all the traces of the process P such that length of the traces are less than or equal to n.

For all closed CSP term, the traces model allows us to calculate the meaning of processes. The meaning of process names (in CSP terms) is given by an environment M. The semantic of the process name is $traces_M(\$p) = M(p)$.

2.2.1.1 The domain of the Traces model and recursive processes

In this section, we discuss some properties of the domain $\mathcal{T}$ which will be useful to give a semantics for recursive processes. We first give the formal definitions of complete lattice, which will also lay mathematical foundations for the thesis.

Definition 2.2: Partial Order. A set P with a binary relation $\leq \subseteq (P \times P)$ is called a partial order $(P, \leq)$ if the relation $\leq$ has the following properties:
(i) *reflexive*: for all $x \in P$, we have $x \leq x$.
(ii) *transitive*: for all $x \in P, y \in P, z \in P$, we have $x \leq y \wedge y \leq z \Longrightarrow x \leq z$.
(iii) *antisymmetric*: for all $x \in P, y \in P$, we have $x \leq y \wedge y \leq x \Longrightarrow x = x$.

Definition 2.3: Complete Lattice.

Let $(P, \leq)$ be a partial order. Let $X \subseteq P$ be a set.

Upper Bound: X has an upper bound $a \in P$ if $x \leq a$ for all $x \in X$.

Least Upper Bound: X has a least upper bound (denoted as $\sqcup X$) a if a is an upper bound and for all upper bounds x, $a \leq x$.

Lower Bound: X has a lower bound $a \in P$ if $a \leq x$ for all $x \in X$.

Greatest Lower Bound: X has a greatest lower bound (denoted as $\sqcap X$) a if a is a lower bound and for all lower bounds x, $x \leq a$.

Lattice: $(P, \leq)$ is a lattice if every finite set $X \subseteq P$ has a least upper bound $\sqcup X$ and a greatest lower bound $\sqcap X$.

Complete Lattice: $(P, \leq)$ is a complete lattice if every $X \subseteq P$ has a least upper bound $\sqcup X$ and a greatest lower bound $\sqcap X$.

The following lemma is useful in proving a partial order is a complete lattice and a proof is given in Lemma A.1.1 of [Ros98].

Lemma 2.4: Let $(P, \leq)$ be a partial order. If $\sqcup X$ exists for every set $X \subseteq P$, then $\sqcap X$ exists for every set $X \subseteq P$.

Lemma 2.5: A partial order (P,$\leq$) is a complete lattice if every $X \subseteq P$ has a least upper bound $\sqcup X$.

Proof. By the lemma 2.4, we know that $\sqcap X$ exists for every subset X of P, if every $X \subseteq P$ has the least upper bound $\sqcup X$. Therefore, every subset X of P has both $\sqcap X$ and $\sqcup X$. Hence $(P, \leq)$ is a complete lattice. □

The reverse direction is also true. Thus, to prove that a partial order is a complete lattice it is sufficient to prove that each subset has a least upper bound or each subset has a greatest lower bound. We give some definitions that will be used to calculate the meaning for recursive processes. To define the notion of complete partial order, we need to define the notion of a directed set.

Definition 2.6: Directed Set. Let $(P, \leq)$ be a partial order. Let $S \subseteq P$ be a set. S is a directed set under a relation $\leq$ if S is non-empty and if for all $x \in S$ and $y \in S$, there exits $z \in S$ such that $x \leq z$ and $y \leq z$

For example, consider the partial order $(\mathcal{N}, \leq)$ and the set of all natural numbers $\mathcal{N}$. Then the set $\mathcal{N}$ is directed, however does not have a least upper bound. Thus, $(\mathcal{N}, \leq)$ is a lattice, however, not complete. The power set of a given set under inclusion relation is a directed set, a lattice and a complete lattice.

Definition 2.7: Complete Partial Order (CPO). $(S, \leq)$ is a complete partial order, if every directed set has a least upper bound.

Definition 2.8: Pointed Complete Partial Order (Pointed CPO). $(S, \leq)$ is a pointed complete partial order, if $(S, \leq)$ is a complete partial order and S has a bottom element $\perp$: for all $x \in S$, $\perp \leq x$.

Definition 2.9: Monotonous. Let f be a function. f is said to be monotonic if $x \leq y$ implies $f(x) \leq f(y)$.

Definition 2.10: Continuous. Let P and Q be two complete partial orders. Let f be a function from P to Q. f is said to be continuous if, whenever $\Delta \subseteq P$ is directed, $\sqcup\{f(x) \mid x \in \Delta\}$ exists and equals $f(\sqcup\Delta)$.

The following lemma follows trivially from the above definitions.

Lemma 2.11: Every complete lattice $(P, \leq)$ is a pointed complete partial order with $\perp = \sqcup\ \emptyset$ or $\perp = \sqcap\ P$.

Lemma 2.12: If f is continuous, then f is monotonic.

Proof. Suppose $x \leq y$. Since f is continuous, $\sqcup\{f(x), f(y)\} = f(y)$. This implies $f(x) \leq f(y)$. $\square$

Lemma 2.13: $(dom(\mathcal{T}), \subseteq)$ is a complete lattice.

Proof. By Lemma 2.5, to prove that a partial order is a complete lattice it is sufficient to prove that each set has a least upper bound.

Suppose Δ be an arbitrary non-empty subset of dom($\mathcal{T}$).
The least upper bound of Δ in dom($\mathcal{T}$) is given by $\bigcup \Delta$.
Let $Y = (\bigcup_{T \in \Delta} T)$. We prove that $Y \in dom(\mathcal{T})$ and Y is the least upper bound of Δ.

First, we prove that Y is the least upper bound of Δ.
If $t \in T$ for some $T \in \Delta$, then $t \in Y$, hence Y is an upper bound.
Suppose $Y1$ be an arbitrary upper bound. We prove that Y is the least upper bound of Δ.
Suppose $t \in Y$ be an arbitrary trace, then, for some T, $t \in T$. Since $Y1$ is an upper bound, $t \in Y1$.
Since $Y1$ is an arbitrary upper bound, Y is the least upper bound.

Now we prove that $Y \in dom(\mathcal{T})$.
Suppose $t \in Y$. Then, for some $T \in \Delta$, $t \in T$. Since T is prefix closed, $s' \in Y$ for all s' such that $t = s' \frown t'$.
Y is non-empty as $\langle\rangle \in Y$ by definition of Y. Thus Y is non-empty and prefix-closed. Thus $\sqcup\Delta = \bigcup \Delta$. $\square$

We have not discussed the trace semantics of recursive processes. The recursive processes can be defined as equations of the form $P = SP(\$P)$ where P is a process name, and SP is a process expression possibly using P. It says that the process P has the same behaviour as the process $SP(\$P)$.

We apply the Tarski fixed-point theorem to find solutions of recursive processes. It guarantees that solutions exits to the equation

$$X = F(X)$$

when F is continuous. The Tarski fixed-point theorem is given below:

Theorem 2.14: Tarski Fixed point theorem [Tar55, Ros98].
If $(P, \leq)$ is a CPO and $F : P \rightarrow P$ is continuous then F has a least fixed point, μF. The least point of F is given by $\mu F = \bigsqcup\{F^m(\bot) \mid 0 \leq m\}$,
where $F^0(\bot) = \bot$ and $F^0(\bot) = F(F^n(\bot))$.

We have the necessary mathematical notions to assign the semantics of recursive processes. If $SP(\$P)$ is a process with a process name P, let $F(.)$ be the semantic function induced by the process $SP(\$P)$. It follows from Theorem 8.2.1 [Ros98] that all the CSP operators are continuous with respect to the subset order relation over the traces model. We give some examples for induced semantic functions.

The meaning for the equation $P = SP(\$P)$ can be defined by iterative unwinding of the recursive definition, starting with the minimal element $\bot$ in the domain as given by the Tarski fixed point theorem. From the below equation, it is clear that $\mu\, P.F(P) = \cup_{n \in \mathcal{N}}\{(F^n(\bot)) \mid 0 \leq n\}$ is the fixed points for the equation $P = SP(\$P)$.

$$\begin{aligned} F(\mu P.F(P)) &= F((\cup_{n \in \mathcal{N}}\{(F^n(\bot)) \mid 0 \leq n\})) \\ &= \cup_{n \in \mathcal{N}}\{(F(F^n(\bot)) \mid 0 \leq n\}) \quad \textit{by continuity} \\ &= \cup_{n \in \mathcal{N}}\{((F^{n+1}(\bot)) \mid 0 \leq n\}) \\ &= \mu P.F(P) \end{aligned}$$

where $F^0(\bot) = \bot$
$F^{n+1}(\bot) = F(F^n(\bot))$.

Thus the semantics of the equation $P = SP(\$P)$ is $\cup_{n \in \mathcal{N}} F^n(\bot)$ where $F(.)$ is the semantic function induced by the process $SP(\$P)$.

Consider for example the recursive process $CLOCK = a \rightarrow \$CLOCK$ in the traces model. The induced semantic function $F(CLOCK)$ of $a \rightarrow \$CLOCK$ is

$$\{\langle\rangle\} \cup \{\langle a \rangle ^\frown t \mid t \in CLOCK\}$$

The successive iterations $F(CLOCK)$ for this definition yields the following

$$\begin{aligned} &F^0(\bot) &&= \bot = \{\langle\rangle\} \\ &F^1(\bot) &&= F(F^0(\bot)) = \{\langle a \rangle, \langle\rangle\} \\ &F^2(\bot) &&= F(F^1(\bot)) = \{\langle a, a \rangle, \langle a \rangle, \langle\rangle\} \\ &F^3(\bot) &&= F(F^2(\bot)) = \{\langle a, a, a \rangle, \langle a, a \rangle, \langle a \rangle, \langle\rangle\} \\ &\vdots \end{aligned}$$

Taking the union of all the values yields the semantics of the process. Thus the semantics of the equation $CLOCK = a \rightarrow \$CLOCK$ is

$$\bigcup\{F^n(\bot) \mid n \in \mathcal{N}\}.$$

A similar construction can be used to find the solution of mutually recursive equations such as

$$P = F(\$P, \$Q)$$

$$Q = G(\$P, \$Q)$$

even in some cases when the number of equations is infinite [RBH81].

We define a few processes which will be used later and have standard names in CSP.

$$RUN_A = ?x : A \rightarrow \$RUN_A$$

The process RUN_A will continuously engage-in the events from A. We define another operator which similarly to RUN_A, but it may non-deterministically terminate any time. It is also defined recursively as follows:

$$CHAOS_A = \texttt{STOP} \sqcap ?x : A \rightarrow \$CHAOS_A.$$

$CHAOS_A$ can perform any events in A except diverge.

Definition 2.15: Divergent. A process P is divergent if P can perform infinite sequence of invisible actions.

2.2.1.2 Traces Refinement

Refinement is a concept central to the design and verification of processes. In CSP, process refinement is a standard technique to compare processes. In the traces model, the process refinement $\sqsubseteq_{\mathcal{T}}$ is defined by reverse containment of trace sets.

Definition 2.16: Refinement Relation ($\sqsubseteq_{\mathcal{R}}$).
Let *Spec* and P be processes. $Spec \sqsubseteq_{\mathcal{T}} P$ if and only if $traces_M(P) \subseteq traces_M(Spec)$

It is read as P is a refinement of *Spec* or *Spec is refined* by P or P *refines Spec* in the model $\mathcal{T}$. It says that all the possible traces of P are also traces of *Spec*. *Spec* can be seen as a specification of the process P and P can be seen as an implementation of the process *Spec*, as P does perform only the traces that are allowed by *Spec*.

Two processes are said to be equivalent if each refines the other. In the traces model, this is defined as below:

Definition 2.17: Equivalence Relation ($=_{\mathcal{T}}$). Let *Spec* and P be processes.
$Spec =_{\mathcal{T}} P$ if and only if $Spec \sqsubseteq_{\mathcal{T}} P$ and $P \sqsubseteq_{\mathcal{T}} Spec$.

It is read as P is traces equivalent with Q. It is interesting to note that the refinement relation $\sqsubseteq_{\mathcal{T}}$ is a partial order. In terms of process algebra, the refinement $Spec \sqsubseteq_{\mathcal{T}} P$ is equivalent to $Spec =_{\mathcal{T}} Spec \sqcap P$. It is proved in the below lemma.

Lemma 2.18: $Spec \sqsubseteq_{\mathcal{T}} P$ *iff* $Spec =_{\mathcal{T}} Spec \sqcap P$.

Proof. $Spec \sqsubseteq_{\mathcal{T}} P \Leftrightarrow traces_M(P) \subseteq traces_M(Spec)$ (by definition $\sqsubseteq_{\mathcal{T}}$)
$\Leftrightarrow traces_M(P) \cup traces_M(Spec) =_{\mathcal{T}} traces_M(Spec)$
$\Leftrightarrow Spec \sqcap P =_{\mathcal{T}} Spec$ (by definition $\sqcap$) □

This says that *Spec* is less deterministic than P. It guarantees that all the behaviours exhibited by the process P should be exhibited by the process *Spec*. Starting from an abstract process, refinement relations helps us to move towards more deterministic processes in each step. Thus $\sqsubseteq_{\mathcal{T}}$ suites well for stepwise refinement from specification to implementation. We can easily observe that $P \sqsubseteq_{\mathcal{T}}$ STOP for all processes P.

It follows from Theorem 8.2.1 of [Ros98] that $\sqsubseteq_{\mathcal{T}}$ is monotonic: let $F(.)$ be a context built from CSP operators and constants; then

$$P \sqsubseteq_{\mathcal{T}} Q \Longrightarrow F(P) \sqsubseteq_{\mathcal{T}} F(Q).$$

$\sqsubseteq_{\mathcal{T}}$ is preserved after applying $F(.)$. This can be proved by induction on structure of the syntax.

2.2.2 The Stable Failures Model

The traces model gives information only on the sequences of events that a process can engage in. This is useful to verify safety properties such as event *fail* does not appear in the traces of a process, event *modified* should appear after event *start* and before event *finished* in any traces of a process, etc. Hence, the traces model gives information such as *nothing bad will ever happen.* Liveness properties such as *good thing will happen* can be specify in the stable failures model. For example, the process $a \rightarrow SKIP$ guarantees that if the environment is prepared to engage in the event a and then terminate, then it can engage in the event a and terminate successfully. However, $a \rightarrow SKIP \sqcap a \rightarrow STOP$ does not guarantee that it can engage in the event a and terminate successfully if the environment is ready to engage in the event a and terminates. The traces model does not identify as both processes have the same traces, but one guarantees that it will terminate successfully, but the other does not guarantee. To capture this kind of property, the stable failures model has been developed.

The stable failures model gives a finer information by distinguishing the above information. A failure of a process is a pair (s, X) that says the process can perform the trace s and reach a state from which it can neither perform any event in X nor any internal event. The set X is called the refusal set; the process cannot perform any event in the set X no matter how long it is offered. The stable failures are the key to distinguish non-deterministic processes. The stable failures model records stable failures and traces of the processes.

In the stable failures model, each process is modelled by a pair (T, F) where $T \subseteq \Sigma^{*\checkmark}$ and $F \subseteq \Sigma^{*\checkmark} \times \mathbb{P}(\Sigma^{\checkmark})$ satisfying the following healthiness conditions:

T1. T is non-empty and prefix closed. This condition is same as the healthiness condition of the the traces model.

T2. $\forall\, s, X . (s, X) \in F \implies s \in T$. This says that all the traces performed by failures should be recorded in the traces component T. This establishes consistency between the traces component and the failures component.

T3. $\forall\, s, X . s ^\frown \langle\checkmark\rangle \in T \implies (s ^\frown \langle\checkmark\rangle, X) \in F$. If a trace $\forall\, s. s ^\frown \langle\checkmark\rangle$ terminates successfully by producing $\checkmark$, then it should refuse all events in $\Sigma^{\checkmark}$ at the stable state after $s ^\frown \langle\checkmark\rangle$.

F2. $\forall\, s, X . (s, X) \in F \wedge Y \subseteq X \implies (s, Y) \in F$. This says in a stable state if a set X is refused, then any subset Y of X should also be refused.

F3. $\forall\, s, X, Y . (s, X) \in F \wedge \forall a \in Y . s ^\frown \langle a\rangle \notin T \implies (s, X \cup Y) \in F$. This says that in a stable state if no event in Y can happen in the next step, then the stable state should also refuse Y besides X.

F4. $\forall\, s . s ^\frown \langle\checkmark\rangle \in T \implies (s, \Sigma) \in F$. This condition says any terminating trace $s ^\frown \langle\checkmark\rangle$ should refuse Σ at the stable state after s.

Definition 2.19: The domain of the stable failures model. The domain of the stable failures model $dom(\mathcal{F})$ is the set of all (T, F) that satisfies the healthiness conditions **T1**, **T2**, **T3**, **F2**, **F3**, and **F4**.

The stable *failures* of a process P can be calculated inductively over the syntax of P as it is done for $traces_M(P)$. The traces component of the stable failures model is same as the traces model.

The semantic definition of failures is given in Figure 2.3. The semantic definition of the timeout and the interrupt operators are not given. The failures of the process SKIP is again two clauses: either $\checkmark$ has happened or not. At the empty trace, it refuses all events other than $\checkmark$ and after successful termination by producing $\checkmark$ it refuses all the events in $\Sigma^{\checkmark}$. The failures of the process STOP are that at the empty traces which refuses everything. The failures of the process DIV have no stable states, hence it has no failures.

$$
\begin{aligned}
\mathit{failures}_M(\mathit{Skip}) &= \{(\langle\rangle, X) \mid X \subseteq \Sigma\} \cup \{(\langle\checkmark\rangle, X) \mid X \subseteq \Sigma^{\checkmark}\}\\
\mathit{failures}_M(\mathit{Stop}) &= \{(\langle\rangle, X) \mid X \subseteq \Sigma^{\checkmark}\}\\
\mathit{failures}_M(\mathit{Div}) &= \emptyset\\
\mathit{failures}_M(a \rightarrow P) &= \{(\langle\rangle, X) \mid a \notin X\}\\
&\quad \cup \{(\langle a\rangle ^\frown t', X) \mid (t', X) \in \mathit{failures}_M(P)\}\\
\mathit{failures}_M(?\, x : A \rightarrow P(x)) &= \{(\langle\rangle, X) \mid A \cap X = \emptyset\}\\
&\quad \cup \{(\langle x\rangle ^\frown t', X) \mid (t', X) \in \mathit{failures}_M(P(x)), x \in A\}\\
\mathit{failures}_M(P \,\Box\, Q) &= \{(\langle\rangle, X) \mid (\langle\rangle, X) \in \mathit{failures}_M(P) \cap \mathit{failures}_M(Q)\}\\
&\quad \cup \{(t, X) \mid (t, X) \in \mathit{failures}_M(P) \cup \mathit{failures}_M(Q), t \neq \langle\rangle\}\\
&\quad \cup \{(\langle\rangle, X) \mid X \subseteq \Sigma,\ \langle\checkmark\rangle \in \mathit{tracesM}(P) \cup \mathit{tracesM}(Q)\}\\
\mathit{failures}_M(P \sqcap Q) &= \mathit{failures}_M(P) \cup \mathit{failures}_M(Q)\\
\mathit{failures}_M((!!\ c : C \bullet P(c)) &= \bigcup\{\mathit{failures}_M(P(c)) \mid c \in C\}\\
\mathit{failures}_M(\mathit{if}\ c\ \mathit{then}\ P\ \mathit{else}\ Q) &= \text{if } c \text{ evaluates to } \mathit{True} \text{ then } \mathit{failures}_M(P) \text{ else } \mathit{failures}_M(Q)\\
\mathit{failures}_M(P \,[\![X]\!]\, Q) &= \{(u, Y \cup Z) \mid Y - (X \cup \{\checkmark\}) = Z - (X \cup \{\checkmark\}),\\
&\quad \exists\, t_1, t_2.\ (t_1, Y) \in \mathit{failures}_M(P), (t_2, Z) \in \mathit{failures}_M(Q),\\
&\quad u \in t_1 \,[\![X]\!]\, t_2\}\\
\mathit{failures}_M(P \setminus X) &= \{(t \setminus X, Y) \mid (t, Y \cup X) \in \mathit{failures}_M(P)\}\\
\mathit{failures}_M(P[[R]]) &= \{(t, X) \mid \exists\, t'.\ (t', t) \in R^*,\ (t', R^{-1}(X)) \in \mathit{failures}_M(P)\}\\
\mathit{failures}_M(P \,;\, Q) &= \{(t_1, X) \mid t_1 \in \Sigma^*,\ (t_1, X \cup \{\checkmark\}) \in \mathit{failures}_M(P)\}\\
&\quad \cup \{(t_1 ^\frown t_2, X) \mid t_1 ^\frown \langle\checkmark\rangle \in \mathit{tracesM}(P),\\
&\qquad (t_2, X) \in \mathit{failures}_M(Q)\}\\
\mathit{failures}_M(P \,\lfloor\, n) &= \{(t, X) \in \mathit{failures}_M(P) \mid |t| < n \vee (\exists\, t'.\ t = t' ^\frown \langle\checkmark\rangle, |t| = n)\}\\
\mathit{failures}_M(\$p) &= \mathtt{snd}(M(p))
\end{aligned}
$$

where $X \subseteq \Sigma$, $C \in \mathit{Choice}(\Sigma)$, $c \in \mathit{Bool}$, $a \in \Sigma$, $\mathtt{snd}(x, y) = y$, $R \in \mathbb{P}(\Sigma \times \Sigma)$, $n \in \mathit{Nat}$, and $p \in \Pi$(set of process names). M is an environment.

Figure 2.3: Semantic clauses for the model $\mathcal{F}$ in CSP$_{\text{TP}}$.

The failures of the process $a \rightarrow P$ have two clauses: at the empty trace, it refuses all the events other than the event a, after engaging in the event a, it can refuse all the refusal set of the process P. A failure of the prefix choice process $?x : A \rightarrow P$ is either the stable state at which it has not performed any event and refuses all the events other than A, or after engaging in an event a from the set A, it refuses refusal set of $P[a/x]$.

The failures of the internal choice $P \sqcap Q$ are union of failures of P and Q. The failures of external choice process $P \,\Box\, Q$ is again two possibilities. Before any events are performed and the choice is resolved, a refusal set should be refused by both the processes P and Q. Once a choice is made, then the process will be behaved as the chosen process. Hence after a choice is resolved or made, all the failures of both the processes P and Q are possible. This makes the external choice and the internal choice distinguish each others.

The stable failure of $P \,[\![A]\!]\, Q$ is a failures $(s, X \cup Y)$ such the trace s comes from the set $t_1 \,[\![A]\!]\, t_2$ where t_1 and t_2 comes from traces in failures of P and Q respectively, and set X and Y comes from refusal set in failures of P and Q respectively. The refusal sets X and Y should also satisfy the condition $X \setminus A^{\checkmark} = Y \setminus A^{\checkmark}$.

The failures $P \setminus X$ is stable if the corresponding state of P refuses the whole of X and hence failures of $P \setminus X$ is $(s, Y \setminus X)$ for each failures $(s, Y \cup X)$ of P. For each failures $(s, R^{-1}(X))$ of $P[[R]]$, the corresponding failures of P has to refuse every single event which renames to the set X. A stable failure of $P \mathbin{;} Q$ is either a failures of P or a non terminating traces of P followed by a failure of Q and the trace in failures of Q should be non-empty.

The failures of the depth restriction process $P \lfloor n$ is similar to the traces model. A failure of the depth restriction operator $P \lfloor n$ is a failures of P such that length of the traces is lesser than n or terminating trace of length n. The meaning of process names (in CSP term) is given by environment M which is a function from process names into the domain of the stable failures model. Hence, the type of semantic function is given by $CSP \rightarrow Environment \rightarrow domain(\mathcal{F})$. The semantic meaning of the process name is given by $failures_M(\$p) = snd(M(p))$, where snd returns the second element of the pair.

We need to assign semantics for recursive processes. A recursive process is defined by the equation $P = SP(\$P)$. The process $SP(\$P)$ can be any process expression possibly involving the process name P. Similar to the trace model, the meaning for the equation $P = SP(\$P)$ can be defined by iterative unwinding of the recursive definition, starting with the minimal process DIV as given by Tarski Fixed point theorem. Thus the semantics for the failures component of the equation $P = SP(\$P)$ is

$$\cup_{n \in \mathcal{N}}(F^n(\bot)).$$

where $F(.)$ is the semantic function induced by the process $SP(\$P)$. The semantic functions of the stable failures model induced by all CSP operators are continuous by subset order relation and it is proven in Theorem 8.4.2 [Ros98].

We look at some examples of CSP in the stable failures model. The process $a \rightarrow Stop \,\square\, b \rightarrow Stop$ is represented as

$$(\{\langle\rangle, \langle a\rangle, \langle b\rangle\}, \{(\langle\rangle, \emptyset), (\langle a\rangle, X), (\langle b\rangle, X) \mid X \subseteq \{a, b, \checkmark\}\}).$$

The process $a \rightarrow Stop \sqcap b \rightarrow Stop$ is represented as

$$(\{\langle\rangle, \langle a\rangle, \langle b\rangle\}, \{(\langle\rangle, Y), (\langle\rangle, Z), (\langle a\rangle, X), (\langle b\rangle, X) \mid X \subseteq \{a, b, \checkmark\}, Y \subseteq \{a, \checkmark\}, Z \subseteq \{b, \checkmark\}\}).$$

This example illustrates that both processes are the same in the traces model, but different in the stable failures model. This is because in the internal choice operator, if the environment offers to communicate the event a, the process $a \rightarrow Stop \sqcap b \rightarrow Stop$ may non-deterministically offer only to communicate the event b. Hence, it refuses the event a after the empty trace $\langle\rangle$. Similarly it refuses the event b, however it cannot refuse both the events $\{a, b\}$ at the empty trace $\langle\rangle$. The stable failures model thus distinguishes the internal choice operator from the external choice operator.

The stable failures refinement is defined similar to the trace refinement by reverse containment.

Definition 2.20: Refinement Relation ($\sqsubseteq_{\mathcal{F}}$). Let $Spec$ and P be processes.
$P \sqsubseteq_{\mathcal{F}} Q$ if and only if $traces_M(Q) \subseteq traces_M(P)$ and $failures_M(Q) \subseteq failures_M(P)$.

It is read as Q refines P or P is refined by Q in the model $\mathcal{F}$. We observe that $P \sqsubseteq_{\mathcal{F}}$ DIV for all processes P. The least-defined process is given by

$$CHAOS_{\Sigma} = (\Sigma^{*\checkmark}, \Sigma^{*\checkmark} \times \mathcal{P}(\Sigma^{\checkmark})).$$

$CHAOS_A$ is the most non-deterministic, divergent-free process with alphabets set A. We can note that $\sqsubseteq_{\mathcal{F}}$ satisfies reflexivity, transitive and antisymmetric properties and hence $\sqsubseteq_{\mathcal{F}}$ is partial order.

The greatest lower bound of the empty set is given by $\top_{\mathcal{F}} = (\{\langle\rangle\}, \emptyset)$ which is the maximal process and represents the denotation for the process `DIV`.

Two processes are said to be equivalent if each refines the other. In the stable failures model, it is defined as below:

Definition 2.21: Equivalence Relation ($=_{\mathcal{F}}$). Let *Spec* and *P* be processes.
Spec $=_{\mathcal{F}}$ *P* if and only if *Spec* $\sqsubseteq_{\mathcal{F}}$ *P* and *P* $\sqsubseteq_{\mathcal{F}}$ *Spec*.

2.3 Isabelle

In this section, we describe the theorem prover Isabelle that is used to implement the tool CSP-Prover. This section gives the information needed for the implementation of the stable revivals model in CSP-Prover. In this section, we first introduce the theorem prover Isabelle and then explain how to use it for our purpose. Secondly, we explain the important theories implemented in CSP-Prover. Finally, we focus on how the syntax and semantics of CSP that have been encoded in CSP-Prover.

2.3.1 Isabelle/HOL-Complex

Isabelle [NPW02] is a generic interactive proof assistant developed at Cambridge University and Technical University of Munich. Isabelle supports wide varieties of logics with a high level of automation. It implements the logics and formalisms as object logics using Isabelle's meta logic. Isabelle's meta logic itself is intuitionist higher order logic. It uses several ideas of Logic of Computable Functions (LCF) [Gor00], so it is based on a small logical core guaranteeing logical correctness. Examples of object logic that are formalised in Isabelle are

- Higher Order Logic (HOL)
- HOL-Complex Extension of HOL with complex numbers.
- Higher-Order Logic of Computable Functions (HOLCF)
- First-Order Logic (FOL) (Many-sorted FOL)
- Zermelo-Fraenkel set theory
- Classical Computational Logic (CCL)
- Logic of Computable Functions (LCF)
- First Order Logic with Proof Terms (FOLP)

Creating a new logic in Isabelle means to define a new syntax for formulae and to give inference rules to perform proofs in the logic. Usually a logic consists of a collection of hierarchy of theories. Isabelle is also a framework for formal specification and verification which has been used in proving the correctness of computer hardware or software and proving properties of computer languages and protocols [Pau98]. Such a system is usually defined in one of the above logic and derived the correctness properties.

Isabelle/HOL [NPW06] is a specialisation of Isabelle for HOL, a polymorphic version of Church's Simple Theory of types. HOL can be seen as a polymorphic typed version of classical set theory. In HOL, typed lambda calculus term can be used as terms. HOL generalises the first order logic by allowing variables which can range over functions and predicates. Isabelle/HOL allows declarative style that means a new concept is introduced by definition. The syntax of HOL is similar to λ-calculus and functional programming. Function application is curried. HOL-Complex [Doc06] is an Isabelle/HOL with extensions of rational, real, complex numbers. These extensions provide real division operation. The real and rational number are dense; moreover, real numbers are complete: every set of reals that is bounded above has a least upper number. CSP-Prover uses the logic HOL-Complex in Isabelle. For the implementing the complete partial order based approach, theories in the logic HOL are sufficient. In CSP-Prover, for implementing the complete metric space based approach, the logic HOL-Complex is used.

2.3.2 Theories in Isabelle

Theories in Isabelle are like modules in programming languages. The most basic theory in Isabelle is `Pure` which contains all the basic elements of Isabelle's meta logic. Isabelle's meta logic has meta-level implication ==> , universal quantifiers $\bigwedge$, and equality ==. Types in Isabelle's meta logic consists of basic types and function types of the form σ ==> τ. The terms are those of typed λ-calculus with the usual type constraints. We write $a :: \sigma$ to mean 'a has type σ'. Another commonly used theory is `Main` which contains a collection of basic theories like arithmetic, lists, sets, etc.

HOL-Complex is based on theories of HOL. It has theories for least upper bound, greatest lower bound, sequence, series, rational number, real positive numbers, limits, continuity and differentiation, finite summation, infinite series, etc. All theories in Isabelle/HOL-Complex are described in [Doc06]. To use HOL-Complex logic, we use the theory `Complex_Main`, not the usual theory `Main` from HOL. Theories in Isabelle are organised hierarchically.

All the theories should be in a file whose name is the same as the theory name followed by .thy. In Isabelle, a theory is a named collection of types, functions, and theorems. The general format of a theory T is

```
theory T
 imports B1 ... Bn
   begin
      declarations
      definitions
      proofs
   end
```

Theories are built on extensions of other theories. Theories can be extended by the commands `import` and `use_thy`. In the above format, theory T is created on the top of $B1, \dots Bn$. Theorems, types and functions in $B1, \dots Bn$ are available in T. To distinguish the same identifiers in the different theories, identifier are qualified over theories like $B1.a$, $T.a$. In the syntax of Isabelle, declarations and definitions are similar to functional languages like Haskell or ML, as Isabelle itself has been implemented in ML.

Isabelle has built-in datatypes like `bool` representing the type of boolean values, `int` representing the type of integers, nat representing the type of natural numbers. Isabelle has Peano-style natural numbers. A new datatype can be created using the keyword datatype. The type nat is a recursive datatype generated by the constructors zero and a successor. It is defined as follows:

```
datatype nat = 0 | Suc nat
```

The datatype nat introduces two constructors 0 and Suc.

Nontrivial types are constructed using type constructors like list, the type of list, set, the type of set, pair(a_1, a_2):: $\tau_1 \times \tau_2$ where each a_i is of type τ_i, tuples that are generated using the pairs. The functions fst and snd are used to extract the individual component of a pair. The function types => represents total functions only. type variables are denoted by 'a, 'b. These form polymorphic types like 'a => 'b.

All functions in Isabelle are total. Functions are declared using the keyword consts.

```
consts
 HC ::  "'a list set => bool"
defs
 HC_def :  "HC F == (EX s .  s :  F & s = [])"
```

In the above examples HC is a function which takes one argument F of type 'a list set and returns bool. HC returns True if the set F has the empty list [], else return False. Functions are declared by keyword consts and defined using the keyword defs. We can combine both the definition and declaration using the constdef.

Types in Isabelle can be created from existing types using the type definition command typedef. It creates a new type from a subset of existing type. Every type in HOL is required to be non-empty. Hence the user is obliged to prove that the newly created type is non-empty by giving a witness. Consider an example to create a type even.

```
consts
 even_check::  "nat => bool"
defs
even_check_def:  "even_check a == a mod 2=0"
typedef even = "{ n .  even_check n}"
apply (rule_tac x ="0" in exI)
apply (simp add:  even_check_def)
done
```

This also creates a set even which is a subset of nat set. It generates the following constant functions and asserts that Rep_even and Abs_even are inverse of each other. We will see some more examples in 2.4.

```
even ::  nat set
Rep_even ::  even ==> nat
Abs_even ::  nat ==> even
```

Types can be grouped by axiomatic type classes. Axiomatic type classes are similar to Haskell's type classes, except that all the instance of the type classes are required to satisfy axioms. Consider for example, axiomatic type class monoid below:

```
consts
 times ::  "'a => 'a => 'a" (infixl "[*]" 70)
 one ::  'a
axclass monoid < type
 assoc:  "(x [*] y) [*] z = x [*] (y [*] z)"
 left_unit:  "one [*] x = x"
 right_unit:  "x [*] one = x"
```

The axiomatic type class monoid is a subclass of the predefined class type which is the class of all HOL types. It has three axioms associativity, left and right unity. We declare nat and int to be an instance of class monoid by command instance

```
defs (overloaded)
 times_nat_def:  "x [*] y == x + (y::nat)"
 unit_nat_def:  "one == 0::nat"
defs (overloaded)
 times_int_def:  "x [*] y == x * (y::int)"
 unit_int_def:  "one == 1::int"
instance nat ::  monoid
 apply (intro_classes)
 by ( unfold times_nat_def unit_nat_def, auto)
instance int ::  monoid
 apply (intro_classes)
 by ( unfold times_int_def unit_int_def, auto)
```

We also have to prove that nat and int satisfies all axioms of monoid. The keyword overload indicates that definition intentionally define the type of given function.

Terms in Isabelle/HOL are formed by applying functions to arguments. Isabelle/HOL also borrows some constructs like conditional expression, evaluation of cases from functional programming to represent terms. The syntax for terms is given below:

- The conditional if-expression is given as `if b then` t_1 `else` t_2 where b is of type texttbool and t_1 and t_2 are of same type.
- The syntax of let-expression is `let x = t in u` where u is a term which may have free variable. Multiple binding are separated by semicolons like `let x=`t_1`;` $\cdots$ `;` $x_n = t_n$.
- The syntax for case-expression is `case e of` $c_1 => e_1 \cdots c_1 => e_1$

Formulae are terms of type *bool* with the usual connectives ~, &, |, ->. Equality is available in the form of the infix function = of polymorphic type `'a => 'a => bool`. Usual quantifiers are available and are written as `ALL x. P`, `EX ! x. P`, and `EX x. P`. `EX ! x.P` means that there exists exactly one x that satisfies P.

Isabelle has three kind of variables: *bound*, *free* and *schematic variables*. Schematic variables have ? as its first character. They are similar to free variables, but can be instantiated by another terms during proof steps.

One of the important concepts in Isabelle is the inductively defined set. In an inductively defined set, for each element of the set we can construct a finite proof of membership using the inference rules given for the set. In Isabelle, we define inductively defined sets using inductive command. In

Isabelle inference rules are called introduction rules defined after the keyword `intros`. We define the set of all odd numbers.

```
consts
 odd ::  "nat set"
inductive "odd"
intros
one:
 "Suc 0:  odd "
step:
 " n :  odd ==> (Suc(Suc n )) :  odd"
```

Isabelle automatically generates and proves introduction, elimination, and induction rules. Introduction rules are

(odd.one)	$Suc\ 0 \in odd$
(odd.step)	$n \in odd ==> Suc(Suc(n)) \in odd.$

The generated elimination rule for odd is `odd.elims`:

```
[| (?a::odd); ?a = Suc (0::nat) ==> ?P::bool;
    ALL n::nat .[|?a=Suc(Suc n); n :  odd |] ==> ?P|] ==> ?P
```

The generated induction rule for odd is `odd.induct`:

```
 [| (?xa::nat) :  odd;
 (?P::nat => bool) (Suc 0); ALL n.  [| n :  odd ; ?P n |]
          ==> ?P (Suc (Suc n)) |] ==> ?P ?xa
```

We can prove that numbers of the form $2 \times k + 1$ are odd by the following lemma:

```
lemma add:  "2*k +1 :  odd"
apply (induct_tac k)
apply (simp add:  odd.one)
by (simp add:  odd.step)
```

We will see more examples on inductively defined sets while discussing CSP-Prover.

Terms and types can be abbreviated using type synonyms and constant definitions respectively. Type synonyms are created using the keyword `types`. This helps us to create understandable code. Consider for example

```
types 'a  sequences  =  ('a  list,nat)
```

The above examples creates a type synonyms called `'a sequences`, a pair consisting of a list and its length.

The primitive recursive functions are defined using the keyword `primrec`.

```
consts
 subtract ::  "nat => nat => nat"
primrec
 "subtract m 0 = m"
 "subtract (Suc m) (Suc n) = subtract m n"
```

The above code defines a recursive function to subtract one natural number from another.

Most of the recursive functions on types that are created using datatypes can be easily represented by `primrec`. To declare arbitrary total functions in Isabelle, we use the keyword `recdef`. It gives full pattern-matching, and recursion need not to involve datatypes. However, the user has to prove that argument to recursive will become smaller and terminate.

```
consts quicksort ::  "nat list => nat list"
recdef quicksort "measure length"
 "quicksort [] = []"
 "quicksort (x#xs) = quicksort(f (%y.  y<=x) xs) @ [x] @
                     quicksort(f (%y.  x<y) xs)"
```

Here the user has to prove that the length of list decreases and terminates. It is defined by the keyword `measure length`. Here `length` denotes the function on `list` which returns size of the list. It recursively sort by separating that the elements lesser than or equal to the first element x and elements greater than x, and then combining all together in order. When the list is empty, the function returns the empty list `[]`.

2.3.2.1 Proof Methods

In Isabelle, proofs can be carried out in two different styles: Linear proofs and Structured proofs. Structured proofs resemble the mathematical style of proof. Isabelle/Isar [Wen06] provides a framework for carrying out structured proofs. Isabelle/Isar allows writing proofs in a concise language, which can be easily understood by humans. More details on Isabelle/Isar are available in [Wen06]. In this thesis, all proofs are done in linear style.

In linear style proof, theorems which need to be proved are represented as the initial goal of a proof. Theorems are proved using *tactics* and *tacticals*. In linear style, a proof is tactic script, consisting of commands that change the states of the proof state. Applying tactics to a goal results in a set of subgoals or the goal is proved. A proof state is a list of subgoals. A proof state G looks as follows:

$$\begin{array}{ll} G & \\ 1. & G_1 \\ \vdots & \\ n-1. & G_{n-1} \\ n. & G_n \end{array}$$

Each G_i has a list of assumptions and one conclusion. The assumptions are the local assumptions for this subgoal. A typical goal is a nested implication of the form

$$\alpha_1 \Rightarrow (\alpha_2 \Rightarrow (\ldots(\alpha_n \Rightarrow \phi)\ldots)).$$

The goal is an abbreviation of the form

$$[\alpha_1;\ \alpha_2;\ \ldots \alpha_n] \Rightarrow \phi.$$

It represents the deduction tree of the proof

$$\frac{\alpha_1 \qquad \alpha_2 \qquad \ldots \qquad \alpha_n}{\phi}$$

Methods change the proof state. Methods can take rules as the parameters. Rules are classified into three types. They are introduction, elimination and destruction rules. We briefly describe the application of a rule with methods. Let R be a rule.

$$\text{R} \quad \frac{P_1 \qquad P_2 \qquad P_3 \ldots ; \; P_n}{Q}$$

- Method `rule` suites well for introduction rules. The method `rule R` unifies Q with the current subgoal, replacing it by n new subgoals which are instances of P_1, P_2 ... and P_n. This is a backward reasoning method where the focus is on decomposing the goal into smaller subgoals. In Isabelle, it looks like

$$[[P_1; \; P2; \; \ldots Pn]] \Rightarrow Q.$$

- Method `erule` suites well for elimination rules. The method `erule R` unifies Q with the current subgoal and simultaneously unifies with some assumption. If unification is successful, it eliminates P_1 and replace the subgoal with new instances of P_2, P_3 ... and P_n. Usually an elimination rule look like

$$[[P_1; \; [[P2; \; \ldots ; \; Pn]] \;\;]] \Rightarrow Q.$$

- Method `drule` is useful for destruction rules. It unifies P_1 with one of the assumptions. if successful, it replaces P_1 with Q and replaces the subgoal with new instances of P_2, P_3 ... and P_n. In this method, the matching assumption is deleted. Usually a destruction rule look like

$$[[P_1; \; [[P2; \; \ldots ; \; Pn]] \;\;]] \Rightarrow Q.$$

- Method `frule` is like drule except that the matching assumption is not deleted.

A goal or subgoals can be proven using backward proof or forward proof. In backward proof, a goal is matched with the conclusion of the rule and the premises of the rule becomes subgoals of the goal. Consider a simple example below:

```
lemma example1 :  " [| A ; B |] ==> A & B"
apply (rule conjI)
apply (assumption)
apply (assumption)
done
```

Here we would like to prove $A \wedge B$ assuming A and B. For the above lemma, initially Isabelle displays one goal as $[| \; A \; ; \; B \; |] \Longrightarrow A \wedge B$.
We apply the backward proof command `apply (rule conjI)` which applies proof method `rule` with conjunction introduction rule `conjI`. The rule looks like

$$[|?P \; ; \; ?Q \; |] \Longrightarrow ?P \wedge ?Q.$$

The question mark ? denotes that P and Q are schematics variable or logical variables. These are instantiated by the terms during the proof process by Isabelle's higher order unification. In the above example, `A` is unified with `?P` and `B` is unified with `?Q`. It matches the conclusion of the subgoal with the rule

$$[|?P \; ; \; ?Q \; |] \Longrightarrow ?P \wedge ?Q,$$

producing two subgoals

$$[| A ; B |] \Longrightarrow A$$

$$[| A ; B |] \Longrightarrow B$$

Then both subgoals are proved by the method `assumption`.

In forward proof, a goal is matched with the premises of rule, creating a new goal from the conclusion of the rule. It means deriving new facts from old ones, thus useful for going from the general term to the specific term. Applying the symmetry of equality through `drule mp` is also a forward step. Usually proof scripts usually consists of both styles of commands.

Isabelle provides a set of tools for automatically proving goals and subgoals called classic reasoners and simplifier. They include

- `simp` which simplifies a subgoal using rewriting and decision procedures like arithmetic decision procedure.
- `simp_all` which is similar to `simp` but tries to prove all subgoals.
- `blast` which proves a subgoal using techniques like tableau methods. This is a powerful classic reasoner.
- `fast` which is similar to `blast` but uses a few steps of heuristic only.
- `auto` which combines implication and classic reasoning to prove all subgoals. It leaves what it cannot prove.
- `force` which is similar to `auto`, but it proves one subgoal only. It either terminates successfully proving a subgoal or terminates by failing to prove a subgoal.
- `clarify` performs obvious steps without splitting the subgoals.

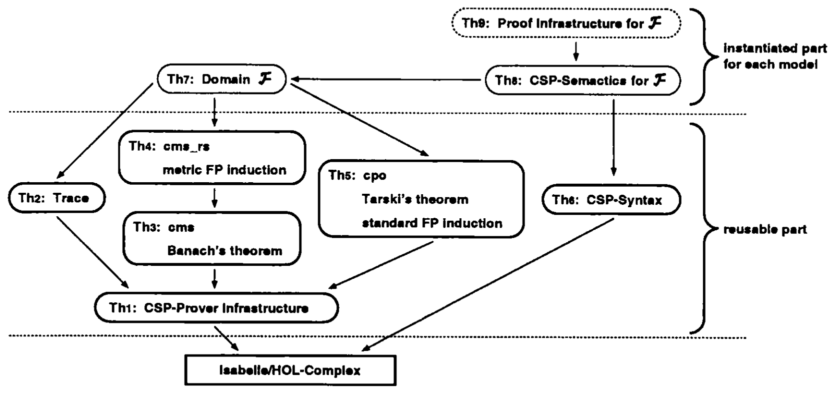
2.4 CSP-Prover

CSP-Prover [IR05] is an interactive proof tool for CSP based on the generic theorem prover Isabelle. It currently fully supports the traces model and the stable failures model. It can be used to prove processes refinement and processes equivalence in these models. Proofs for processes refinement can be exploited using three different strategies:

- Based on semantic proof.
- Based on syntactical proof.
- Based on semi-automatic syntactical proof.

A number of interesting properties have been proven using CSP-Prover in the traces model and the stable failures model. It has been extended by a framework for the deadlock-analysis of networks.

Figure 2.4 shows the architecture of CSP-Prover. CSP-Prover follows a generic approach: individual CSP models can be instantiated separately. It consists of three major packages : `CSP`, `CSP_T` and `CSP_F`. The package `CSP` contains a reusable package which independent of CSP models. `CSP_T` contains theories related to the traces model. `CSP_F` contains theories related to the stable failures



Th9: Proof Infrastructure for $\mathcal{F}$
instantiated part for each model
Th7: Domain $\mathcal{F}$
Th8: CSP-Semactics for $\mathcal{F}$
Th4: cms_rs metric FP induction
Th5: cpo Tarski's theorem standard FP induction
Th2: Trace
Th6: CSP-Syntax
Th3: cms Banach's theorem
reusable part
Th1: CSP-Prover Infrastructure
Isabelle/HOL-Complex

Figure 2.4: The theory map of CSP-Prover instantiated with the stable-failures model $\mathcal{F}$.

model. The packages `CSP_T` and `CSP_F` can be instantiated separately. Both are implemented on theories in `CSP`. In this section, we explain the reusable part of CSP-Prover.

2.4.1 Reusable Part of CSP-Prover

The package CSP, the reusable part of CSP-Prover, contains important theories for Tarski's fixed point theorem and the standard fixed point induction rule based on Complete Partial Orders (CPO), and Banach's fixed point theorem and the metric fixed point induction rule based on Complete Metric Space (CMS). Other important theories in the reusable part are the theory of traces, the theory for the hiding operator, the theory of pairs, the theory for the parallel operator, the theory for the renaming operator, the theory for the sequential operator. In this section, we explain the above theories in detail.

2.4.1.1 The Theory 'Traces.thy' in CSP-Prover

In the theory of traces, a new polymorphic type `'a event` is created on a polymorphic type `'a` and a distinguished element using the following type definitional command

```
datatype 'a event = Ev 'a | Tick
```

The distinguish element `Tick` is added to the polymorphic type `'a`. It is a standard technique to add a value to any existing datatype. In terms of CSP, the constructor `Ev 'a` represents elements of the alphabets set Σ and the constructor `Tick` represents the termination symbol $\checkmark$. All the theories in CSP-Prover are build on this polymorphic type `'a event`.

The datatype `'a trace` is created on `'a event list` using the following command

```
typedef'a trace = "{l::('a event list).  Tick ~:  set(butlast l)}"
```

The expression on the right hand side checks whether `Tick` appears in the set of all the elements of the list except the last element. This is done using function `butlast l` which returns the tail of the list l. `set` is a polymorphic function that converts elements in the list l in to sets. Each trace is a list of events and the event $\checkmark$ may appear only as the last event in the list. It is then proved that the new

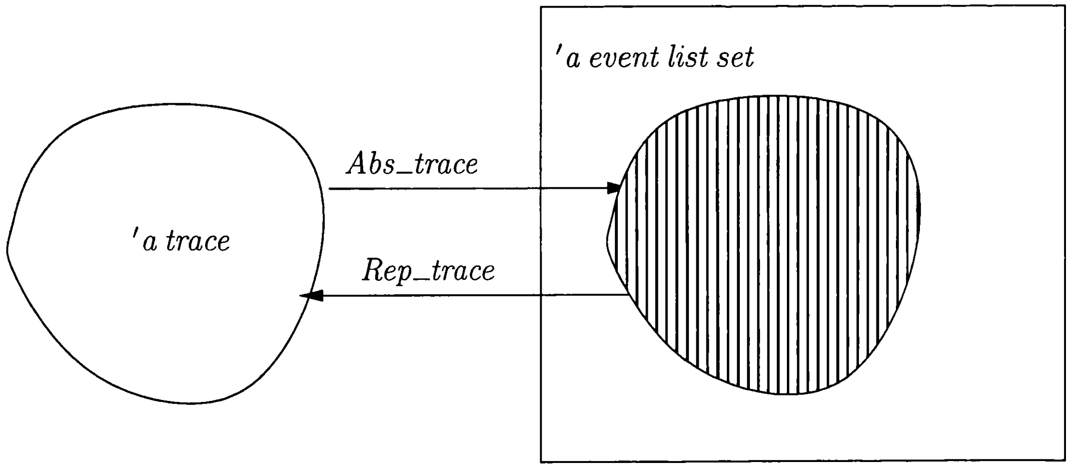


Figure 2.5: Trace representation

datatype is non-empty by giving a witness `[]` to Isabelle. The datatype `'a trace` is a subset of type `'a event list set`.

Isabelle creates the following three constants to manipulate between the representations:

```
trace ::   'a event list set
Rep_trace ::  'a trace ==> 'a event list
Abs_trace ::  'a event list ==> 'a trace
```

It is pictorially represented as shown in the Figure 2.5. Isabelle construction also asserts that `Rep_trace` is surjective on the subset of type `'a event list set` and asserts that `Rep_trace` and `Abs_trace` are inverse of each other. It is shown by the following equations:

```
Rep_trace               Rep_trace x ∈ trace
Rep_trace_inverse       Abs_trace (Rep_trace x) = x
Abs_trace_inverse       x ∈ trace ⟹ Rep_trace (Abs_trace(x)) = x
```

Isabelle creates other useful lemmas like injectivity of `Rep_trace` and `Abs_trace`:

```
(Rep_trace_inject) (Rep_trace x = Rep_trace y) = (x=y)
(Abs_trace_inject) [| x ∈ trace; y ∈ trace |] ⟹
                                  (Abs_trace x = Abs_trace y) =
(x=y)
```

To manipulate and prove properties on level of traces, we raise the level of abstraction by defining required functions on `trace` for each analogous functions in the `list` via `Rep_trace` and `Abs_trace`. The following functions and predicates are defined:

```
consts
 appt ::   "'a trace => 'a trace => 'a trace" (infixr "b 65)
 nilt ::   "'a trace" ("<>")
 sett ::   "'a trace => 'a event set"
 lengtht ::  "'a trace => nat"
 noTick ::  "'a trace => bool"
 hdt ::  "'a trace => 'a event"
 tlt ::  "'a trace => 'a trace"
 lastt ::  "'a trace => 'a event"
 butlastt ::  "'a trace => 'a trace"

defs
 nilt_def :   "<> == Abs_trace []"
 sett_def :   "sett s == set (Rep_trace s)"
 lengtht_def:   "lengtht s == length (Rep_trace s)"
 noTick_def :   "noTick s == (Tick ~:  sett s)"
 hdt_def :   "hdt s == hd (Rep_trace s)"
 tlt_def :   "tlt s == Abs_trace (tl (Rep_trace s))"
 lastt_def :   "lastt s == last (Rep_trace s)"
 butlastt_def :   "butlastt s == Abs_trace (butlast (Rep_trace s))"
```

The concatenation of two traces is defined `appt_def`. The traces s and t are first converted into list representation using `Rep_trace` s and `Rep_trace` t respectively and then perform the concentration of those using list concatenations operator `@` and then convert it back to its trace representation using `Abs_trace`. Syntactic sugar in Isabelle allows that ^^ can be used instead of `appt`. The annotation `infixr` means that ^^ associated to the right and it can be used as infix notation as s ^^ t. The empty trace $\langle\rangle$ is represented as the empty list [] in list representation and it is defined by `nilt`. `sett s` gives the events in the trace `s`. To find `sett s`, we convert the trace `s` into its list representations of the trace `s` using `Rep_trace` and find the elements in the list using `set`.

`NoTick s` checks whether the event `Tick` appears in the traces s or not using `sett`. The initial event of the trace s is given by `hdt s`. It is calculated by first converting the traces `s` into its list representation `Rep_trace s` and then taking the first element in the list using `hd`. Similarly the last event in the traces s is defined by using `lastt_def`. To find the tail of the trace `s`, its corresponding tail of list is found using `tl (Rep_trace s)` and then convert it back it to trace using `Abs_trace`. Similarly, we find `butlastt s` which returns the same trace s except the last event of the trace `s` is removed.

To work solely on `trace`, sufficient properties of `trace` have been proved in the theory of traces.

2.4.1.2 The Theory 'Prefix.thy' in CSP-Prover

The theory of prefix extends the theory of traces:

```
theory Prefix
imports Trace
```

A trace t is a prefix of a trace s, if there exist a trace u such that $t \frown u = s$. A set of traces T is prefix-closed if and only if for each $s \in T$, if t is a prefix of s, then $t \in T$. This is defined as follows:

```
consts
 prefix ::  "'a trace => 'a trace => bool"
 prefix_closed ::  "('a trace set) => bool"
defs
prefix_def:
 "prefix s t == (EX u.  t = s ^u & (noTick s | u = <>))"
prefix_closed_def :
 "prefix_closed T == ALL s t.  ((t :  T & prefix s t) --> s :  T)
```

CSP-Prover follows the definitional approach which means "taking an existing logic for granted, the new objects are represented in terms of existing concepts, and the desired properties are derived from the definitions within the system" [BW99]. This is a simple good example for conservative extension in CSP-Prover. The following lemma has been proved to establish this:

```
lemma prefix_closed_iff:
 "[| t :  T ; prefix s t ; prefix_closed T |] ==> s :  T"
```

The proof directly follows from the definition of prefix and prefix closed. Mathematically it represents

$$\forall t, T, s \;.\; t \in T \wedge \mathit{prefix}\ s\ t \wedge \mathit{prefix\_closed}\ T \Longrightarrow s \in T.$$

The following lemmas have been proved in this theory:

$$\forall s. \mathit{prefix}\ s\ s$$

It asserts that every trace is a prefix of itself.

$$\forall s\ t.(\mathit{Notick}\ s \vee t = \langle\rangle) \Longrightarrow \mathit{prefix}\ s\ (s \frown t)$$

This says that s is a prefix of $s \frown t$ provided t is tickfree or $\checkmark$ does not appear in s.

$$\forall s\ t.((\mathit{NoTick}\ t \vee u = \langle\rangle) \wedge \mathit{prefix}\ s\ t) \Longrightarrow \mathit{prefix}\ s\ (t \frown u)$$

This says s is a prefix of $t \frown u$ provided t is tickfree or u is $\langle\rangle$.

$$\forall s.\mathit{prefix}\ \langle\rangle\ s$$

says that the empty trace is a prefix of every trace. For more other properties, we refer to the theory file Prefix.thy.

2.4.1.3 The Theory 'CPO.thy' in CSP-Prover

The theory for complete partial order contains Tarski fixed point theorem 2.14 and continuity definitions. Many useful lemmas on continuity have been proved in this theory. In this theory, four axiomatic type classes are defined. Isabelle/HOL has a type class `ord` in theory `Main`. `ord` is the type class of all types for which an ordering relation $\leq$ is defined. `ord` has two relations $<=$ and $<$. The functions `mono`, `min` and `max`, and the `LEAST` operator are defined over `ord`. The first axiomatic class is given below:

```
axclass  bot0 < order
consts   Bot ::  " 'a::bot0 " (* Bottom *)
```

The above creates a new axiomatic subclass called `bot0` which has a function `Bot` of type $a :: bot0$. It fixes an element of type as the bottom element.

```
axclass bot < bot0
bottom_bot : "Bot <= (x::'a::bot0)
```

The above code creates another new axiomatic called `bot` from `bot0`. `bot` is a subclass of `bot0`. The newly created class has an axiom which requires that `Bot` to be less than all other elements in `bot0`. This establishes that there is the minimum element in the datatype. An axiomatic type class `cpo` is created as a subclasses of `order` where the following axiom needs to hold. The new axiom says that every directed set has the least upper bound.

```
axclass cpo < order
complete_cpo :  "(directed (X::'a::order set)) ==> X hasLUB"
```

Another axiomatic type class `cpo` with minimum element called `cpo_bot` is created as a subclasses of cpo and order using the below code:

```
axclass cpo_bot < cpo , bot
```

The definition of continuous function is defined as below:

```
constdef
 continuous ::  "('a::cpo => 'b::cpo) => bool”
 continuous_def :  "continuous f == ALL X. directed X -- >
                   ((f ` X) hasLUB & LUB (f ` X) = f (LUB X))"
```

The function f is continuous if for every directed set X, $\{f(x) \mid x \in X\}$ has a least upper bound, and the least upper bound of is equal to $f(\sqcup X)$. It is then proved that

$$\forall X.directedX \implies \exists\, x.(x = \sqcup X \wedge f(x) = \sqcup\{f(x).x \in X\})$$

Other important lemmas have also been proved in this theory are follows

$$\forall\, f.continuous\, f \implies mono\, f$$

It says if f is continuous, then f is monotonic as proved in Lemma 2.12.

$$\forall\, f\; g.continuous\, f \wedge continuous\; g \implies continuous\;\; f\; o\; g.$$

It says the composition of two continuous functions is also a continuous function.

$$\forall f\; X.continuous\, f \wedge directed\, X \implies directed\{f(x) \mid x \in X\}.$$

This says that if f is continuous and X is directed set, then $\{f(x) \mid x \in X\}$ is also a directed set. The important Tarski Fixed point theorem is proved in this theory.

$$\forall f.continuous\, f \implies f \text{ has a least fixed point.}$$

2.4.1.4 Production of CPO and CPO pair

The theory `CPO_prod` has lemmas to show that infinite products of CPOs are a CPO. The theory `CPO_pair` provides theorems to show that a pair of CPOs is a CPO. It also has a continuity proof for functions on these spaces. Here we discuss only the theory on `CPO_pair` as a theory on `CPO_prod` is similar. It is done by making the operator $*$ (binary product) to be an instance of axiomatic type classes declared in CPO.

```
instance * :: (bot, bot) bot0
by (intro_classes)
```

There are no axioms in the axiomatic type classes `bot0`, hence the proof obligation is completed by command
`intro_classes`. We overload element `Bot` by `(Bot,Bot)`. This is done by the following command:

```
defs (overloaded)
pair_Bot_def :  "Bot == (Bot, Bot)"
```

Next, the operator $*$ is declared as an instance of axiomatic type class `bot`. Here `bot` has an axiom. Hence, it is neccassary to prove that `bot` satisfies the axiom using the command `by (simp add: pair_Bot)`.

```
instance * :: (bot, bot) bot
apply (intro_classes)
by (simp add:  pair_Bot)
```

The command `by (simp add:  pair_Bot)` discharges remaining proof obligation. Then it is proven that a pair of CPOs is also a CPO. This is done by making the operator $*$ an instance of axiomatic type class `cpo`. Here `bot` has an axiom, which says every directed set has a least upper bound. The following does the above job:

```
instance * :: (cpo,cpo) cpo
apply (intro_classes)
by (simp add:  pair_cpo_lm del:  split_paired_Ex)
```

Similarly, $*$ is declared is an instance of axiomatic type class `cpo_bot` which is an axiomatic type classes for pointed CPO.

2.4.1.5 Syntax of CSP in CSP-Prover

In this section, we discuss how the syntax of CSP has been encoded in CSP-Prover. CSP-Prover follows a deep encoding; hence, the syntax is defined in HOL. In CSP-Prover, the syntax of CSP is defined in the reusable packages and the semantic function is defined in the individual packages for each models separately. The syntax of CSP is defined as a separate datatype `('p,'a) proc.` where $'p$ is the type of process name and $'a$ is the type of communication alphabet Σ.

```
datatype ('p,'a) proc =
 STOP
|SKIP
|DIV
|Act_prefix    " 'a"      " ('p,'a) proc "           (" (1_ /-> _) " [150,80] 80)
|Ext_pre_choice   " 'a set"     " 'a => ('p,'a) proc "
                            (" (1?  :_ /-> _) " [900,80] 80)
|Ext_choice              " ('p,'a) proc "      " ('p,'a) proc "
                            (" (1_ /[+] _) " [72,73] 72)
|Int_choice              " ('p,'a) proc "      " ('p,'a) proc "
                             ("(1_ /| | _)" [64,65] 64)
|Rep_int_choice          " 'a sets_nats "      " 'a aset_anat => ('p,'a) proc "
                            (" (1!!  :_ ..  /_) " [900,68] 68)
|Interrupt              " ('p,'a) proc "      " ('p,'a) proc "
                            (" (1_ /[\ _) " [72,73] 72)
|Timeout                " ('p,'a) proc "      " ('p,'a) proc "
                            (" (1_ /[> _) " [73,74] 73)
|IF                      " bool "      " ('p,'a) proc "       " ('p,'a) proc "
                        (" (0IF _ /THEN _ /ELSE _) " [900,60,60] 88)
|Parallel                " ('p,'a) proc " " 'a set " " ('p,'a) proc "
                            (" (1_ /|[_]| _) " [76,0,77] 76)
|Hiding                " ('p,'a) proc "      " 'a set "               (" (1_ /-- _) "
[84,85] 84)
|Renaming              " ('p,'a) proc "      " ('a * 'a) set "
                            ( "(1_ /[[_]]) " [84,0] 84)
|Seq_compo             " ('p,'a) proc "      " ('p,'a) proc "
                            (" (1_ /;; _) " [79,78] 78)
|Depth_rest            " ('p,'a) proc "      " nat "
                            (" (1_ /|.  _) " [84,900] 84)
|Proc_name             " 'p "                 (" $_ " [900] 90)
```

The type of index sets in the replicated internal choice is defined by following command

```
types
'a sets nats = " ('a set set, nat set)   sum "
'a aset anat = " ('a set, nat)   sum "
```

An index-set is disjoint union of subset of subsets of communications or a subset of natural numbers. Disjoint union set is defined in the theory `Infra_fun` as `sum`.

```
datatype ('a,'b) sum = type1 " 'a " | type2 " 'b "
```

Later, sending and receiving for the prefix operator is defined as

```
consts
Send_prefix:: "('x => 'a) => 'x => ('p,'a) proc => ('p,'a) proc"
                                ("(1_ !  _ /-> _)" [900,1000,80] 80)
Nondet_send_prefix:: "('x => 'a) => 'x set =>
                                ('x => ('p,'a) proc) => ('p,'a) proc"
Rec_prefix:: "('x => 'a) => 'x set =>
                                ('x => ('p,'a) proc) => ('p,'a) proc"
defs
Send_prefix_def:
                                "a !  x -> P == a x -> P"
Nondet_send_prefix_def:   "Nondet_send_prefix f X Pf
            == !  :(f ' X) -> (%x.  (Pf ((inv f) x)))"
Rec_prefix_def:
"Rec_prefix f X Pf == ?  :(f ' X) -> (%x.  (Pf ((inv f) x)))"
```

By defining the syntax as a separate datatype, we can quantify over syntactic structures of CSP. For example, we can prove the formula,

$$\forall p \in ('p,'a)\ proc\ .\ p \,\Box\, p = p\ .$$

2.4.1.6 Theory of the Renaming Operator 'Trace_ren.thy'

The renaming operator in CSP takes a set of relations. The set of all traces generated by this renaming relations is defined using inductive definition, so it takes a set of relation as parameters. The inductive definition is a function that yields sets. The inductive defined set of renaming relation is as follows:

```
consts
 renx ::   "('a * 'b) set => ('a trace * 'b trace) set"
inductive
 "renx R"
intros
 renx_nil:   "(<>, <>)  :   renx R"
 renx_nil:   "(<>, <>)  :   renx R"
 renx_Tick:   "(<Tick>, <Tick>)  :   renx R"
 renx_Ev:   "[| (s, t)  :   renx R ; (a, b)  :   R |]
            ==> (<Ev a> ^^ s, <Ev b> ^^ t)  :   renx R"
```

The definition consists of three rules. The first two rules are obvious: renx_nil says $\langle\rangle$ is renamed to $\langle\rangle$ and renx_Tick says $\langle\checkmark\rangle$ is renamed to $\langle\checkmark\rangle$. The last rule renx_Ev says that if $(s, b) \in R$ and $(s, t) \in renx\ R$ then $(\langle Ev\ a\rangle \frown s, \langle Ev\ b\rangle \frown t) \in renx\ R$

The above inductive definition generates the least closed set of traces that satisfies the above three definitions. Isabelle also proves useful theorems about it. These theorems include introduction rule, and elimination rule. Isabelle generates introduction rules with the names specified in the declaration.

$(\langle\rangle, \langle\rangle) \in renx\ R$	renx.renx_nil
$(\checkmark, \checkmark) \in renx\ R$	renx.renx_Tick
$(s, t) \in renx\ R \wedge (a, b) \in R) \Rightarrow (\langle Ev\ a\rangle \frown s, \langle Ev\ b\rangle \frown t) \in renx\ R$	renx.renx_Ev

It produces an induction rule ($renx.induct$), and an elimination rule ($renx.elims$).

The renaming of a trace is defined by the following function:

```
consts
 ren_tr :: " 'a trace => ('a * 'b) set => 'b trace => bool "
                         ( "(_ [[_]]* _) " [1000,0,1000] 1000)
defs
 ren_tr_def: " s [[r]]* t == (( s, t) :  renx R) "
```

The code says s is renamed by t by the renaming r if $(s, t) \in renx\ R$. It also says $[[R]]*$ is syntax sugar for renaming a trace. Hence, $[[R]]^*$ is the smallest set satisfying the following inference rules:

$$\begin{gathered} \mathit{True} \Rightarrow (\langle\rangle, \langle\rangle) \in [[R]]^* \\ \mathit{True} \Rightarrow (\langle\checkmark\rangle, \langle\checkmark\rangle) \in [[R]]^* \\ (a, b) \in R \wedge (t, t') \in [[R]]^* \Rightarrow (a ^\frown t,\ b ^\frown t') \in [[R]]^* \end{gathered}$$

The inverse relation of R is defined as follows:

```
defs
ren_inv_def:
 "[[R]]inv X == ea.  EX eb :  X. ea = Tick ∧ eb = Tick |
(EX a b.  (a,b):R ∧ ea = Ev a ∧ eb = Ev b)"
```

2.4.1.7 Theory of the Hiding Operator 'Trace_hide.thy'

Similar to the theory of the renaming operator, the theory of the hiding operator is defined using inductive definitions. The hiding operator $P \setminus X$ takes a set as input. The code is shown below:

```
consts
 hidex :: "'a set => ('a trace * 'a trace) set"
inductive "hidex X"
intros
 hidex_nil:     "(⟨⟩, ⟨⟩) : hidex X"
 hidex_Tick:     "(⟨Tick⟩, ⟨Tick⟩) : hidex X"
 hidex_in:          "[| (s, t) : hidex X ; a : X |] ==> (⟨Eva⟩ ^^ s, t) : hidex X"
 hidex_notin:      "[| (s, t) : hidex X ; a ~: X |] ==> (⟨Eva⟩ ^^ s, ⟨Eva⟩ ^^ t) : hidex X"
```

The first rule says that hiding of the empty trace by any set X is the empty trace, denoted as $\langle\rangle \setminus X = \langle\rangle$. The second rule says that hiding the trace $\langle\checkmark\rangle$ by any set X results in the same trace $\langle\checkmark\rangle$. The third rule says that hiding a trace which has an element in X results in removing the element from the trace. The last rule says that hiding an element which is not in the set X results the same trace. The inductive definition generates a set of pairs such that the first element intuitive denotes the trace before hiding, and the second element denotes the traces after hiding the elements in X. The hiding a trace is formally defined as

```
defs
 hide_tr_def :   "s --tr X == THE t.  (s, t) :  hidex X"
```

Then existences and uniqueness are proved by the following lemmas

```
lemma hidex_exists:   "EX t.  (s, t) :  hidex X"
lemma hidex_unique:
"[| (s, t) :  hidex X ; (s, u) :  hidex X |] ==> t = u"
```

2.4.1.8 Theory of the Parallel Operator 'Trace_par.thy'

The theory of the parallel operator is also defined based on inductively definition set. Given traces s and t, $s[[X]]t$ denotes the set of all traces such that event in X must be synchronised for the traces s and t. It also takes an synchronisation events set X and returns a set of triples. The code of inductive definition is given below:

```
consts
 parx ::  "'a set => ('a trace * 'a trace * 'a trace) set"
```

```
1   inductive "parx X"
2   intros
3   parx_nil_nil:  "(⟨⟩,⟨⟩,⟨⟩) :  parx X"
4   parx_Tick_Tick:  "(<Tick>, <Tick>, <Tick>) :  parx X"
5   parx_Ev_nil:  "[| (u, s, <>) :  parx X ; a ~:  X |]
6            ==> (<Ev a> ^^u, <Ev a> ^^ s, <>) :  parx X"
7   parx_nil_Ev:  "[| (u, <>, t) :  parx X ; a ~:  X |]
8           ==> (<Ev a> ^^ u, <>, <Ev a> ^^ t) :  parx X"
9   parx_Ev_sync "[| (u, s, t) :  parx X ; a :  X |]
10           ==> (<Ev a> ^^ u, <Ev a> ^^ s, <Ev a> ^^ t) :  parx X"
11  parx_Ev_left:  "[| (u, s, t) :  parx X ; a ~:  X |]
12           ==> (<Ev a> ^^ u, <Ev a> ^^ s, t) :  parx X"
13  parx_Ev_right:  "[| (u, s, t) :  parx X ; a ~:  X |]
14            ==> (<Ev a> ^^ u, s, <Ev a> ^^ t) :  parx X"
```

It generates a set of triples (u, s, t) for an input X. The trace u is the result of parallel composition of the traces s and t. The first rule says that composition of two empty traces $\langle\rangle$ results in the empty trace $\langle\rangle$. Line number 7 says that if $(u, s, t) \in parx\ X$ and $a \in X$, then
$(\langle Ev\ a\rangle \frown u, \langle Ev\ a\rangle \frown s, \langle Ev\ a\rangle \frown t) \in parx\ X$.

Chapter 3

Motivation for the Stable Revivals Model

Contents

In this section, we discuss the motivation for the stable revivals model. The stable revivals model was developed in response to work by Fournet et al [FHRR04] on conformance relations in CCS. On the application side, the stable revivals model suites well for reasoning about responsiveness and stuck-freeness. Responsiveness and stuck-freeness are important properties in Component Based Systems Design. In Section 3.1, we focus at Component Based Systems Design and its relation with responsiveness and stuck-freeness properties. In Section 3.2 and Section 3.3, we give the definitions of responsiveness and stuck-freeness.

3.1 Component Based Systems

Component Based Systems Design is an approach in which existing components are plugged in order to build complex software. Components are reusable software programs that can be developed separately and assembled easily to create complex applications.

A widely accepted definition is given by Szyperski [Szy02]:

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

The definition says components can be composed. To compose, components should provide well defined interfaces. These interfaces act as a contract between components and environment. The properties of components are specified in component specifications. A Component is a self-contained description. A component specification has the following four aspects [LO01]:

1. Signature of component. This deals with the syntactical aspects of components. It is usually provided by specification of interface. In Component Object Model (COM) [Box97], Distributed Component Object Model (DCOM) [Tha99], Common Object Request Broker Architecture (CORBA) [Gro08], interfaces are usually defined in their own Interface Definition Language (IDL). This also forms the basic of interaction.

2. Semantics of component. It deals with the semantic aspects of components. It is important for automatic testing of components and to check whether the client code satisfies pre-conditions. The Object Constraint language (OCL) in the Unified Modeling Language (UML) [UML08] can specify semantic part of components. The signature and semantics of components characterise the components ability.

3. Packaging or Interaction. This deals with aspects of how components can be composed with each other. This discusses the protocol level properties of different communicating components or interoperability of components. Responsiveness and stuck-freeness discuss interactions of components in a larger system.

4. Quality attributes(illities). This deals with non-functional properties of components such as security, reliability, and performances.

The motivation for using components based systems is that they reduce the overall system development costs. Components can be bought or reused from already existing code instead of being developed from scratch. Individual components in the distributed system will need an assurance that the interacting components will not behave in an undesirable manner. Components need a proof of correctness before communicating with other components. Responsiveness and stuck-freeness are properties of this kind.

3.2 Definition of Responsiveness

A component Q is responsive to another component P if the component Q will not cause the component P to deadlock and furthermore, no refinement of Q causes any refinement of P to deadlock. This is related to the question whether two processes deadlock when put in parallel. However, it is different in that we require that a specific process P is not itself blocked by a plug-in Q when it could have otherwise have progressed [RRS04, RRS06]. [RRS06] defines RESPONDSTOLIVE and RESPONDSTO where the first fails to capture responsiveness. RESPONDSTO captures the intuition behind the notion that we are intended to have. More properties and examples on these definitions are given in [RRS06]. Below we explain examples similar to [RRS06].

3.2.1 First Definition on Responsiveness

We give the formal definition of responsiveness in CSP over the model $\mathcal{F}$. We assume all processes to be divergence-free. Each process P has its own alphabet $\alpha(P)$, but it might communicate only some of its events.

Let $J = \alpha(P) \cap \alpha(Q)$. Let $J^{\checkmark} = J \cup \{\checkmark\}$ be the set of shared events between process P and process Q with the termination signal $\checkmark$ included. We can consider process P as the requesting process which requires process Q to respond in a non-blocking manner. The definition RESPONDSTOLIVE [RRS06] is given below:

Definition 3.1: RESPONDSTOLIVE. A process Q RESPONDSTOLIVE process P on A for $A \subseteq J^{\checkmark}$ if for every trace s such that $(s, A) \in \mathit{failures}_M(P \mathbin{|[} J \mathbin{]|} Q) \implies (s, A) \in \mathit{failures}_M(P)$ holds. We say Q RESPONDSTOLIVE P if $A = J^{\checkmark}$.

Q RESPONDSTOLIVE P on A means that if $P \mathbin{|[} J \mathbin{]|} Q$ can reach a state after executing the trace s and A is refused at that state, then the process P will also refuse the event set A after the same trace s. We illustrate RESPONDSTOLIVE by the below examples where $J = \alpha(P) = \alpha(Q) = \{req, rep1, rep2\}$.

Example 3.2:

$$P = req \rightarrow (rep1 \rightarrow \mathtt{SKIP} \;\square\; rep2 \rightarrow \mathtt{SKIP})$$
$$Q = req \rightarrow rep1 \rightarrow \mathtt{SKIP}$$

For the above example, $\mathit{failures}_M$ of P and Q are

$$\begin{aligned}
\mathit{failures}_M(P) &= \{(\langle\rangle, X) \mid X \subseteq \{rep1, rep2, \checkmark\}\} \cup \{(\langle req\rangle, X) \mid X \subseteq \{req, \checkmark\}\} \cup \\
&\quad \{(\langle req, rep1\rangle, X), (\langle req, rep2\rangle, X) \mid X \subseteq \{req, rep1, rep2\}\} \cup \\
&\quad \{(\langle req, rep1, \checkmark\rangle, X), (\langle req, rep2, \checkmark\rangle, X) \mid X \subseteq \{req, rep1, rep2, \checkmark\}\} \\
\mathit{failures}_M(Q) &= \{(\langle\rangle, X) \mid X \subseteq \{rep1, rep2, \checkmark\}\} \cup \{(\langle req\rangle, X) \mid X \subseteq \{req, rep2, \checkmark\}\} \cup \\
&\quad \{(\langle req, rep1\rangle, X) \mid X \subseteq \{req, rep1, rep2\}\} \cup \\
&\quad \{(\langle req, rep1, \checkmark\rangle, X) \mid X \subseteq \{req, rep1, rep2, \checkmark\}\} \\
\mathit{failures}_M(P \mathbin{|[} J \mathbin{]|} Q) &= \mathit{failures}_M(Q).
\end{aligned}$$

Consider the above example when $A = \{req, rep1, rep2\}$. The event set A is refused by $P \mathbin{|[} J \mathbin{]|} Q$ after the two traces $\langle req, rep1\rangle$ and $\langle req, rep1, \checkmark\rangle$. P also refuses A after the same traces. Thus for every traces s, we have

$$(s, A) \in \mathit{failures}_M(P \mathbin{|[} J \mathbin{]|} Q) \implies (s, A) \in \mathit{failures}_M(P)$$

Hence, Q RESPONDSTOLIVE P on A.

Consider the above example when $A = \{req, rep2\}$. A is refused by $P \mathbin{|[} J \mathbin{]|} Q$ after the traces in $\{\langle\rangle, \langle req\rangle, \langle req, rep1\rangle, \langle req, rep1, \checkmark\rangle\}$, but A is not refused by P after the trace $\langle req\rangle$. Hence, Q RESPONDSTOLIVE P on A is not true.

Consider the below examples where $J = \alpha(P) = \alpha(Q) = A = \{rep\}$.

Example 3.3:

$$P = (rep \rightarrow \mathtt{SKIP}) \sqcap \mathtt{SKIP}$$
$$Q = rep \rightarrow \mathtt{SKIP}$$

$$\begin{aligned}
\mathit{failures}_M(P) &= \{(\langle\rangle, X) \mid X \subseteq \{\checkmark, rep\}\} \cup \{(\langle rep\rangle, X) \mid X \subseteq \{rep\}\} \\
&\quad \{(\langle \checkmark\rangle, X), (\langle rep, \checkmark\rangle, X) \mid X \subseteq \{rep, \checkmark\}\} \\
\mathit{failures}_M(Q) &= \{(\langle\rangle, X) \mid X \subseteq \{\checkmark\}\} \cup \{(\langle rep\rangle, X) \mid X \subseteq \{rep\}\} \cup \\
&\quad \{(\langle rep, \checkmark\rangle, X) \mid X \subseteq \{rep, \checkmark\}\} \\
\mathit{failures}_M(P \mathbin{|[} \{rep\} \mathbin{]|} Q) &= \{(\langle\rangle, X) \mid X \subseteq \{\checkmark, rep\}\} \cup \mathit{failures}_M(Q)
\end{aligned}$$

$\{rep\}$ is refused by $P \,[\![J]\!]\, Q$ after the traces $\langle rep\rangle$ and $\langle rep, \checkmark\rangle$ when $\{rep\}$ is also refused by P after the same traces. Hence, Q RESPONDSTOLIVE P on $\{rep\}$. But P RESPONDSTOLIVE Q is not true as $(\langle\rangle, \{rep\}) \in failures_M(P \,[\![\{rep\}]\!]\, Q)$ and $(\langle\rangle, \{rep\}) \notin failures_M(Q)$. Here P might non-deterministically terminate and can refuse to engage in $\{rep\}$ after $\langle\rangle$.

The definition RESPONDSTOLIVE captures the desired behaviour in most of the cases, but Example 3.4 shows that it sometimes does not capture the intended behaviour.

Example 3.4:

$$P = rep \rightarrow \mathtt{STOP}$$
$$Q = (rep \rightarrow \mathtt{STOP}) \sqcap \mathtt{STOP}$$

$$\begin{aligned}
&failures_M(P) && = \{(\langle\rangle, X) \mid X \subseteq \{\checkmark\}\} \cup \{(\langle rep\rangle, X) \mid X \subseteq \{rep, \checkmark\}\}\\
&failures_M(Q) && = \{(\langle\rangle, X) \mid X \subseteq \{rep, \checkmark\}\} \cup failures_M(P)\\
&failures_M(P \,[\![J]\!]\, Q) = failures_M(Q)
\end{aligned}$$

The examples satisfies Q RESPONDSTOLIVE P, but Q does not engage in rep when P engages in rep. This clearly violates the required definition of responsiveness namely that Q does not cause P to deadlock. Consider another two processes P' and Q' which are refinements of P and Q respectively in the model $\mathcal{F}$.

Example 3.5:

$$P' = rep \rightarrow \mathtt{SKIP}$$
$$Q' = \mathtt{STOP}$$

$$\begin{aligned}
&failures_M(P') && = \{(\langle\rangle, X) \mid X \subseteq \{\checkmark\}\} \cup \{(\langle rep\rangle, X) \mid X \subseteq \{rep, \checkmark\}\}\\
&failures_M(Q') && = \{(\langle\rangle, X) \mid X \subseteq \{rep, \checkmark\}\}\\
&failures_M(P' \,[\![J]\!]\, Q') = failures_M(Q')
\end{aligned}$$

In the above example P' RESPONDSTOLIVE Q' is not true as $(\langle\rangle, \{rep\}) \in failures_M(P' \,[\![J]\!]\, Q')$ and $(\langle\rangle, \{rep\}) \notin failures_M(P')$.

Definition 3.6: Refinement-closed. Let ϕ be a relation on specifications. ϕ is refinement-closed if and only if, given $P\phi Q$ then for all P', Q' such that $P \sqsubseteq P'$ and $Q \sqsubseteq Q'$, it is the case that $P\phi Q \implies P'\phi Q'$.

Example 3.4 and Example 3.5 show that P RESPONDSTOLIVE Q is not refinement-closed. RESPONDSTOLIVE has been defined as a predicate, and to verify it in a refinement checker like FDR [Lim07], we need to represent the predicate as a refinement relation. In [RRS04], RESPONDSTOLIVE has been formulated as machine-checkable assertions suitable for verification in tools like FDR. We briefly explain it in Appendix A.3.

3.2.2 The Second Definition on Responsiveness

The previous examples show that the definition RESPONDSTOLIVE captures some false positives. Q RESPONDSTOLIVE P on A means that for every traces s either $(s, A) \notin failures_M(P[\![J]\!]Q)$ or $(s, A) \in failures_M(P)$ holds. In the first case, $(s, A) \notin failures_M(P[\![J]\!]Q)$ means that the parallel

process do not block each other on A. If $(s, A) \in \mathit{failures}_M(P)$ means that a blocking on A has been caused by process P. In the second case if Q is considered as not blocking even if Q is the source of blocking. Example 3.4 illustrates this problem clearly. It can be improved by adding more tighter constraints like refinement-closed on RESPONDSTOLIVE. [RRS04] gave another definition of responsiveness RESPONDSTO which captures the required behaviour. In [RRS04], it is also proved that P RESPONDSTO Q if and only if P' RESPONDSTOLIVE Q' for all the refinements P' and Q' of P and Q respectively. We define some notations which will be used in the definition:
Let $\mathit{initials}(P)$ is the set of all initial events in which P may engage ;
P/s is the process which behaves as P would after execution of trace s;
$s \restriction A$ is the sub-sequence of s formed by restricting s to elements of set A.

Definition 3.7: RESPONDSTO. A Process Q RESPONDSTO process P if and only if for all
$s \in (\alpha P \cap \alpha Q)^*$, $X \subseteq J^{\checkmark}$, such that
$(s \restriction \alpha P, X) \in \mathit{failures}_M(P) \wedge (J^{\checkmark} \cap \mathit{initials}(P/s)) - X \neq \{\} \Longrightarrow$
$(s \restriction \alpha Q, (J^{\checkmark} \cap \mathit{initials}(P/s)) - X) \notin \mathit{failures}_M(Q)$.

For any failures $(s \restriction \alpha P, X) \in \mathit{failures}_M(P)$, the set $(J^{\checkmark} \cap \mathit{initials}(P/s)) - X \neq \{\}$ describes a joint events in which P wants Q's participation.

Example 3.4 does not satisfy the RESPONDSTO as $(\langle\rangle, \{\})$ is a failure of P with $J^{\checkmark} \cap \mathit{initials}(P/s)) - \{\} = \{\mathit{rep}\}$ which satisfies the left hand side and fails in right hand side as $\{\mathit{rep}\} \in \mathit{failures}_M(Q)$.

RESPONDSTO is the weakest refinement-closed strengthening of RESPONDSTOLIVE [RRS04]. In [RRS06], both RESPONDSTO and RESPONDSTOLIVE have been formulated as machine-checkable assertions in the CSP model checker, FDR. We will give the definition for RESPONDSTO in the stable revivals model and illustrate its difference with the stable failures model in Chapter 4.

3.3 Definition of Stuckness

In this section, we give the definition of stuckness. The stuckness property is defined in CCS. To introduce stuckness, we briefly give the syntax and semantics of CCS in Appendix A.5.

3.3.1 Stuckness, and Conformance

In CCS, two processes communicate with each other through the rule $Com3$. Handshake is an atomic communication in which a data value is sent by one process and received by another simultaneously. It is the basis for communication in CCS. The transition diagram for handshaking communication is given below:

$$\mathit{Res}\ \cfrac{\mathit{Com}_3\ \cfrac{\mathit{Act}\ \cfrac{}{\alpha.P \xrightarrow{\alpha} P} \qquad \mathit{Act}\ \cfrac{}{\overline{\alpha}.Q \xrightarrow{\overline{\alpha}} Q}}{\alpha.P \mid \overline{\alpha}.Q \xrightarrow{\tau} P \mid Q}}{(\alpha.P \mid \overline{\alpha}.Q) \setminus \{\alpha\} \xrightarrow{\tau} (P \mid Q) \setminus \{\alpha\}}$$

Hence it is interesting to check whether the process $P \setminus X$ terminates successfully or not. Informally for $(P \mid Q) \setminus X$, checking whether interactions between P and Q succeed or not for events in X.

In other words, it is to check for sending messages that are never received or waiting for receiving messages that are never sent. The following definition captures stuck-freeness of a process.

Definition 3.8: Stuck. A process P is *stuck* on X, if $P \setminus X$ is an end-state, and $P \xrightarrow{\lambda}$ for some $\lambda \in X$. We refer to such λ as a residual action

Definition 3.9: Stuck-free process. A process is called *stuck-free* on X, if there is no P' and $\tilde{\alpha}$ such that $P \xrightarrow{\tilde{\alpha}} P'$ with $set(\tilde{\alpha}) \cap X = \emptyset$, and P' is stuck on set $\tilde{\alpha}$, where $set(\tilde{\alpha})$ returns the element in the sequences $\tilde{\alpha}$ as a set.

The restriction operator of CCS in the definition of stuckness and the condition $set(\tilde{\alpha}) \cap X = \emptyset$ in the definition of stuck-free process implies that only internal events can happen on names in X.

In Definition 3.8, a process $(P \mid Q) \setminus X$ is stuck, if $P \mid Q \xrightarrow{\lambda}$ for some $\lambda \in X$ means that interaction between P and Q do not succeed. One of the processes from P and Q is waiting for its co-action $\overline{\lambda}$ to happen, but the other process never engages in $\overline{\lambda}$.

Let $init(P) = \{\alpha \mid P \xrightarrow{\alpha}\}$, and let $\mathcal{L}^{(1)}$ denote the singleton sets of $\mathcal{L}$ together with the empty set, $\mathcal{L}^{(\infty)} = \{\{\lambda\} \mid \lambda \in \mathcal{L}\} \cup \{\emptyset\}$.

Definition 3.10: Refusal. If X is a subset of $\mathcal{L}$, we say that P refuses X if and only if P is stable and $init(P) \cap \overline{X} = \emptyset$. We say that P can refuse X if and only if there exists P' such that $P \xrightarrow{\tau^*} P'$ and P' refuses X.

Definition 3.11: Readiness. If $Y \in \mathcal{L}^{(1)}$, we say that P is ready on Y, if and only if P is stable and $\lambda \in Y$ implies $P \xrightarrow{\lambda}$. Any stable process is trivially ready on $\emptyset$.

Definition 3.12: Ready Refusal. If $X \subseteq \mathcal{L}$ and $Y \in \mathcal{L}^{(1)}$, we say that P can refuse X while ready on Y if and only if P can refuse X from a state that is ready on Y, *i.e.*, there exists P' such that $P \xrightarrow{\tau^*} P'$, P' refuses X, and P' is ready on Y.

Definition 3.13: Conformance Relation. A binary relation $\mathcal{R}$ on processes is called conformance relation if and only if, whenever $P \mathcal{R} Q$, then the following conditions hold:
$C1$. If $P \xrightarrow{\tau^*} Q'$ then there exists Q' such that $Q \xrightarrow{\tau^*} P'$ and $P'\mathcal{R}Q'$.
$C2$. If P can refuse X while ready on Y, then Q can refuse X while ready on Y.

Condition $C2$ is very similar to the stable failures refinement. We define composition of two conformance relations $\mathcal{R}_1$ and $\mathcal{R}_2$ as $\mathcal{R}_1 o \mathcal{R}_2 = \{(P, Q) \mid \exists R.(P, R) \in \mathcal{R}_1 \wedge (R, Q) \in \mathcal{R}_1$. In [FHRR04], the following lemma has been proved:

Lemma 3.14: Let $\{\mathcal{R}_i\}_{i \in I}$ be a family of conformance relations. Then
1. The relation $\cup_{i \in I} \mathcal{R}_i$ is a conformance relation.
2. For any $i, j \in I$, the relation $\mathcal{R}_i o \mathcal{R}_j$ is a conformance relation.
3. The identity relation on processes is a conformance relation

Using the above lemma 3.14, we can define the largest conformance relation $\leq$ by taking the union of all conformance relation.

Definition 3.15: Stuck-free conformance $\leq$. The largest conformance relations is referred to as (stuck-free) conformance and is denoted $\leq$. We write $P \leq Q$ for $(P, Q) \in \leq$, and we say that P conforms to Q.

In [FHRR04], it is proved that the conformance relation $\leq$ is reflexive and transitive. As an example for conformance relation we consider $(P = a.\mathbf{0} \mid b.\mathbf{0}) \leq (Q = (a.b.\mathbf{0})\#(b.a.\mathbf{0}))$. Considered over the alphabet of names $\{a, b\}$. The process $(a.b.\mathbf{0})\#(b.a.\mathbf{0})$ refuses $\{a, b, \overline{b}\}$ from the state it is ready on $\{a\}$, refuses $\{b, a, \overline{a}\}$ from the state it is ready on $\{b\}$. After engaging in a, P refuses $\{a, \overline{a}, b\}$ accepts $\{b\}$, and after engaging in b, P refuses $\{b, \overline{b}, a\}$ accepts $\{a\}$. The process $a.b.\mathbf{0} + b.a.\mathbf{0}$ refuses $\{a, b\}$ and is ready on both $\{a\}$ and $\{b\}$. After engaging in a, P refuses $\{a, \overline{a}, b\}$ accepts $\{b\}$, and after engaging in b, Q refuses $\{b, \overline{b}, a\}$ accepts $\{a\}$. Similarly after engaging in a, Q refuses $\{a, \overline{a}, b\}$ accepts $\{b\}$, and after engaging in b, Q refuses $\{b, \overline{b}, a\}$ accepts $\{a\}$. Hence it satisfies $C1$ and $C2$.

But the reverse direction $(Q = (a.b.\mathbf{0})\#(b.a.\mathbf{0})) \not\leq (P = a.\mathbf{0} \mid b.\mathbf{0})$ does not hold as $C2$ fails. P refuses $\{a, b, \overline{b}\}$ from the state that it ready on $\{a\}$, but Q does not refuse $\{a, b, \overline{b}\}$ from the state it is ready on $\{a\}$.

Let $\mathcal{C}$ range over contexts, which are process expression with a hole (written []) in them:

$$\mathcal{C} ::= [] \mid (P \mid []) \mid ([] \mid P) \mid (\alpha.[] + M) \mid ([]) \setminus X$$

$\mathcal{C}$[P] denotes the agent that arises by substituting P for the hole in $\mathcal{C}$. In [FHRR04], the following three theorems have been proved.

Theorem 3.16: Pre congruence. $P \leq Q$ implies $\mathcal{C}(P) \leq \mathcal{C}(Q)$.

It is similar to the refinement relation is monotonic in CSP. It says the stuck-free conformance is preserved by all contexts.

Theorem 3.17: Preservation. $P \leq Q$ and P is not stuck-free on X, then Q is not stuck-free on X.

It says stuck-free conformance preserves the ability to get stuck.

Theorem 3.18: Substitutability. Assume $P \leq Q$. Then $\mathcal{C}$[Q] stuck free on X implies $\mathcal{C}$[P] stuck free on X.

These properties are similar to that the stable revivals model is fully abstract with respect to condition responsiveness and stuck-freeness properties [Ros07].

3.4 Relation between Responsiveness and Stuck-freeness

The property RESPONDSTO is motivated by the need for proper operations of plug-in components in Component Based Systems Design. We captured RESPONDSTOLIVE and RESPONDSTO in the stable failures model. On the other hand, the stuck-freeness property is motivated to check whether in an interaction between two processes there is no unreceived message present or not. Stuckness is similar to deadlock in the stable failures model, but is more discriminative than CSP deadlock [FHRR04].

Stuckness is directly observable in a labelled transition system. In the stable revivals model, we can capture stuckness property similarly to CCS. Conformance in CCS is useful for reasoning about both deadlock and unreceived message in asynchronous systems. Stuck-free conformance extends the stable failures relations by requiring that a stable state fails to engaging events in set X while being ready to engaging in event a.

RESPONDSTO is an asymmetric property. Two interacting processes which synchronise on their entire alphabets satisfy RESPONDSTO in both directions if and only if they are stuck-free on the entire alphabets [Ros07].

Chapter 4

The Stable Revivals Model

Contents

In Chapter 3, we looked at the motivations for the stable revivals model, especially responsiveness and stuckness properties. We have also seen relationships between the responsiveness and stuckness. The stable revivals model is created to capture properties like responsiveness and stuckness in a more precise way than that it is possible in the stable failures model. In this chapter, we discuss the stable revivals model of CSP and its semantics. We will also present the semantics of the stable revivals model. Finally, we present a definition of responsiveness and stuckness in the stable revivals model. It turns out that a new healthiness condition needs to be added in the definition of the domain and it is explained in Section 4.1.1.

4.1 The domain of the Stable Revivals Model

The stable revivals model is a *finite observation model.* Thus the stable revivals model records finite traces and other information detectable in a finite amount of time, including refusal events observable on some stable states. We give the definition of the finite observation model following [Ros07] in terms of the behaviours that can be observed on processes:
(i) behaviours of a process take a finite amount of time to observe,
(ii) behaviours only record things that can be seen on a single interaction with the process - they are linear, and
(iii) behaviours are restricted to what can reasonably be observed of a standard labelled transition system in which, from one state, one cannot "see ahead" to a range of behaviours that can follow its initial actions.

In the stable revivals model [Ros07], Σ is finite and each process P is identified by a triple (T, D, R), where

- $T \subseteq \Sigma^{*\checkmark}$ consists of all P's finite traces which P can execute.
- $D \subseteq \Sigma^*$ consists of all traces after which P can possibly deadlock. Since a successfully terminated trace (a trace with ending with $\checkmark$) does not lead to a deadlock, $\checkmark$ does not appear in the deadlock traces.
- $R \subseteq (\Sigma^* \times \mathcal{P}(\ \Sigma)\times\ \Sigma)$ consists of all revivals of P. A revival is a triple in which the first element is a trace of the process P, the second element is the refusal set in a stable state after the given trace and the third element is a non-tick event that the process P can accept in the stable state after the trace. The third element is called "reviving event". Thus in a stable state of revival, the process does not engage in $\checkmark$ or any internal events. Each revival (s, X, a) represents that process P can perform trace s, stably refuse X and then it can accept the event a. Revivals are similar to failures in the stable failures model, but extended with a reviving event.

Each triple (T, D, R) should satisfy the following healthiness conditions:

T1. T is nonempty and prefix-closed; i.e. if $s \frown t \in T$, then $s \in T$.

D1. $D \subseteq T$. i.e. every trace that leads to deadlock is a possible trace.

R1. $(s, X, a) \in R \Longrightarrow s \frown \langle a\rangle \in T$. This says that every trace implied by a revival should be in T. This establishes that the traces and revivals components of a process are consistent.

R2. $(s, X, a) \in R \wedge Y \subseteq X \Longrightarrow (s, Y, a) \in R$. This says in a revival, all subsets of the stable refusal event X should also be refused. The refusal set of any revivals is subset closed.

R3. $(s, X, a) \in R \wedge b \in \Sigma \Longrightarrow ((s, X, a) \in R \vee (s, X \cup \{b\}, a) \in R)$. This says that any event b should appear in the refusal set of the revival or in the acceptance set.

We also consider the following condition introduced in [RRS06]:

RRS05. $(s, X, a) \in R \Longrightarrow a \notin X$. This says the revival event a is not allowed to appear in the refusal set X.

Definition 4.1: The Domain of the model $\mathcal{R}$.
The domain of the stable revivals model $dom(\mathcal{R})$ is defined to be the set of all triples (T, D, R) satisfying the above healthiness conditions. Mathematically,

$$dom(\mathcal{R}) = \{(T, D, R) \mid (T, D, R) \text{ satisfies } \mathbf{T1, D1, R1, R2, R3} \text{ and } \mathbf{RRS05.}\}$$

For example, $(\langle\rangle, \emptyset, \emptyset)$ is in the domain of $\mathcal{R}$ as it satisfies all the healthiness conditions. We give some more examples of the process representation after giving the semantics for each operators in CSP$_{\text{TP}}$.

4.1.1 Inclusion of the healthiness condition RRS05

Condition **RRS05** has been separately given in [RRS06]. However it is not given in the definition of the domain of the model $\mathcal{R}$. Here we include it explicitly in the definition of the domain as it is necessary to prove the type correctness of the hiding operator in CSP-Prover in Page 86. We will focus on this later when proving the type correctness of the hiding operator in Chapter 6.

The full abstraction property [Ros98] in CSP asserts two things:

Expressibility: The semantic function of the model is surjective. Each element in the domain is represented by some processes.

Distinctiveness: There should be some contexts or criteria for distinguishing processes. It is proved in [Ros07] that the stable revivals model is fully abstract with respect to stuckness and responsiveness.

Had we not included Condition **RRS05** in the definition of the domain, then we would have had the following triple in the domain.

Let $\Sigma = \{a\}$. Then

$$C = (\{\langle\rangle, \langle a\rangle\}, \{\}, \{(\langle\rangle, \{a\}, a), (\langle\rangle, \{\}, a)\})$$

C fulfils all the healthiness conditions except **RRS05**, but there is no process which represents the above triple. Hence, it would be a counter example in expressibility of the full abstraction.

4.1.2 The Stable Revivals Refinement

The notion of refinement within the stable revivals model is defined by component-wise set inclusion similar to the stable failures model.

Definition 4.2: Refinement Relation ($\sqsubseteq_{\mathcal{R}}$).
Let (T_P, D_P, R_P) be the denotational value of P and (T_Q, D_Q, R_Q) be the denotational value of Q in the model $\mathcal{R}$. $P \sqsubseteq_{\mathcal{R}} Q$ if and only if $(T_Q \subseteq T_P)$, $(D_Q \subseteq D_P)$, and $(R_Q \subseteq R_P)$.

$P \sqsubseteq_{\mathcal{R}} Q$ is read as Q refines P by the model $\mathcal{R}$. This says that the traces of Q are contained in the traces of P, the deadlocks of Q are contained in the deadlocks of P, and the revivals of Q are contained in the revivals of P. In the stable failures model, the refinement guarantees that the refined process will never refuse events that were not refusable by the specification. But, in the revivals, the refinement guarantees the same as in the failures, but also asserting that refined process will accept only the reviving events that are allowed by the specification. In the next section, we prove the domain is a CPO on the component-wise set inclusion relation. Two processes are equal in the model $\mathcal{R}$ if each refines other.

Definition 4.3: Equivalence Relation ($=_{\mathcal{R}}$).
Let P and Q be processes. $Q =_{\mathcal{R}} P$ if and only if $Q \sqsubseteq_{\mathcal{R}} P$ and $P \sqsubseteq_{\mathcal{R}} Q$.

It is read as P is equivalent with Q in $\mathcal{R}$. We can easily show that $(dom(\mathcal{R}), \subseteq)$ is a partial order as the component-wise set inclusion satisfies reflexivity, transitivity and antisymmetric properties.

4.1.3 The Domain of the Stable Revivals Model is a Pointed Complete Partial Order

Here first we prove that $(dom(\mathcal{R}), \subseteq)$ is a complete lattice. We know that $(dom(\mathcal{R}), \subseteq)$ is a complete lattice implies $(dom(\mathcal{R}), \subseteq)$ is a pointed complete partial order. Hence we prove the stronger case that $(dom(\mathcal{R}), \subseteq)$ is a complete lattice and $(\{\langle\rangle\}, \emptyset, \emptyset)$ is a bottom element. $(\{\langle\rangle\}, \emptyset, \emptyset)$ trivially satisfies the conditions **T1**, **D1**, **R1-R3** and **RRS05**. It is clear that $\{\langle\rangle\} \subseteq T$, $\emptyset \subseteq D$, and $\emptyset \subseteq R$, for all $(T, D, R) \in dom(\mathcal{R})$.

Theorem 4.4: $(dom(\mathcal{R}), \subseteq)$ is a complete lattice.

Proof. By Lemma 2.5, to prove that a partial order is a complete lattice it is sufficient to prove that each set has a least upper bound.

Suppose $\Delta \subseteq dom(\mathcal{R})$ and Δ is non-empty.
We define $Y = (\bigcup_{(T,D,R) \in \Delta} T, \bigcup_{(T,D,R) \in \Delta} D, \bigcup_{(T,D,R) \in \Delta} R)$

We prove that $Y \in dom(\mathcal{R})$ and Y is the least upper bound of Δ.

Suppose $t \in T$ where $(T, D, R) \in Y$, then there exists some $X = (T', D', R') \in \Delta$ such that T' is non-empty and prefix-closed, hence $s' \in T$ for all s' such that $t = s' \frown t'$. T is prefix-closed and non-empty. Hence Y satisfies **T1**.

Suppose $s \in D$ where $(T, D, R) \in Y$, then there exists some $X = (T', D', R') \in \Delta$. We know that (T', D', R') is healthy by assumption, this means that $D' \subseteq T'$. By assumption we know that $s \in T'$, therefore by the definition of Y, $s \in T$. Hence Y satisfies **D1**.

Suppose $(s, X, a) \in R$ where $(T, D, R) \in Y$, we know that there is some X such that $X = (T', D', R') \in \Delta$ and $(s, X, a) \in R'$. By assumption, we know that $s \frown \langle a \rangle \in R'$, therefore by definition of Y, $s \frown \langle a \rangle \in R$. Hence Y satisfies **T1**. The proof follows similarly for other conditions.

We now prove that Y is an upper bound of Δ. We prove this only for the revivals component as the proof for other components is similar. Suppose $r \in R$ where $X \in \Delta$ and $X = (T, D, R)$, then by definition of Y, we know that there exists $r \in R_Y$ such that $Y = (T_Y, D_Y, R_Y)$. Hence Y is an upper bound.

Now we prove that for all upper bounds $Y' = (T_{Y'}, D_{Y'}, R_{Y'})$, that $Y = (T_Y, D_Y, R_Y)$ is the least upper bound: i.e., $(T_Y \subseteq T_{Y'})$, $(D_Y \subseteq D_{Y'})$, and $(R_Y \subseteq D_{Y'})$. We prove this only for revivals component.
Suppose r is an arbitrary revival in R_Y where $Y = (T_Y, D_Y, R_Y)$, then by the definition of Y, we know that there exists X such that $X = (T', D', R') \in \Delta$ and $r \in R'$. As $Y' = (T_{Y'}, D_{Y'}, R_{Y'})$ is an upper bound of Δ, $r \in R_{Y'}$. Since r is an arbitrary, we know that for all $r \in R_Y$, we have $r \in R'_Y$. □

4.2 Semantics of The Stable Revivals Model

In this section, we discuss the semantic meaning for the CSP operators in the stable revivals model. The traces component in $\mathcal{R}$ is identical to the traces model presented as in Chapter 2. Hence in this section, we focus mainly on the semantic functions for the deadlock component and the revivals component.

- $$\begin{aligned} traces_M(\texttt{SKIP}) &= \{\langle\rangle, \langle\checkmark\rangle\} \\ deadlocks_M(\texttt{SKIP}) &= \emptyset \\ revivals_M(\texttt{SKIP}) &= \emptyset \end{aligned}$$

  Since the terminating process SKIP does not contribute anything to deadlock traces, the deadlock component of SKIP is the empty set. The only event that the process SKIP can produce is $\checkmark$. As other processes can only observe the event $\checkmark$, there is no need for other processes to agree on it. Since reviving events cannot be ✓, the revivals component of SKIP is also the empty set.

- $$\begin{aligned} traces_M(\mathtt{STOP}) &= \{\langle\rangle\} \\ deadlocks_M(\mathtt{STOP}) &= \{\langle\rangle\} \\ revivals_M(\mathtt{STOP}) &= \emptyset \end{aligned}$$

Since the non-terminating process STOP contributes to deadlock trace, the deadlock component of STOP has the empty trace $\langle\rangle$. The revivals component of SKIP is the empty set as STOP does not perform any events.

- $$\begin{aligned} traces_M(\mathtt{DIV}) &= \{\langle\rangle\} \\ deadlocks_M(\mathtt{DIV}) &= \emptyset \\ revivals_M(\mathtt{DIV}) &= \emptyset \end{aligned}$$

DIV engages in internal events continuously and does not engaging in any external events. Hence both the deadlock and revival component of DIV are the empty set.

- $$\begin{aligned} traces_M(a \rightarrow P) &= \{\langle\rangle\} \cup \{\langle a\rangle \frown t' \mid t' \in traces_M(P)\} \\ deadlocks_M(a \rightarrow P) &= \{\langle a\rangle \frown t' \mid t' \in deadlocks_M(P)\} \\ revivals_M(a \rightarrow P) &= \{(\langle\rangle, X, a) \mid a \notin X, X \subseteq \Sigma\} \cup \\ &\quad \{(\langle a\rangle \frown t', X, b) \mid (t', X, b) \in revivals_M(P)\} \end{aligned}$$

The process $a \longrightarrow P$ engages in the event a and then behaves as the process P. The deadlock component is similar to the traces component, but it does not have the empty trace $\langle\rangle$. The revivals of the process $a \longrightarrow P$ has two clauses: at the empty trace $\langle\rangle$, it refuses all events different from a, while ready to accept the event a. After engaging in the event a, it refuses all the refusal of P while accepting reviving events of P.

- $$\begin{aligned} traces_M(?\,x : A \rightarrow P(x)) &= \{\langle\rangle\} \cup \{\langle x\rangle \frown t' \mid t' \in traces_M(P(x)), x \in A\} \\ deadlocks_M(?\,x : A \rightarrow P(x)) &= \{\langle x\rangle \frown t' \mid t' \in deadlocks_M(P(x)), x \in A\} \\ revivals_M(?\,x : A \rightarrow P(x)) &= \{(\langle\rangle, X, a) \mid A \cap X = \emptyset, a \in A\} \cup \\ &\quad \{(\langle x\rangle \frown t', X, b) \mid (t', X, b) \in revivals_M(P(x)), x \in A\} \end{aligned}$$

The deadlocks of $?\ x : A \longrightarrow P(x)$ are $\langle\, x \,\rangle$ concatenated with the deadlocks of $P(x)$. A revival of the prefix choice operator $?\ x : A \longrightarrow P(x)$ is either at the empty trace it refuses all the events other than the events in A, but ready to engage in the events in A or after engaging in an event from A, it refuses refusal sets of $P[a/x]$, but ready on reviving events of $P[a/x]$.

- $$\begin{aligned} traces_M(P \sqcap Q) &= traces_M(P) \cup traces_M(Q) \\ deadlocks_M(P \sqcap Q) &= deadlocks_M(P) \cup deadlocks_M(Q) \\ revivals_M(P \sqcap Q) &= revivals_M(P) \cup revivals_M(Q) \end{aligned}$$

A deadlocks of $P \sqcap Q$ is either a deadlocks of P or a deadlocks of Q. Similarly a revivals of $P \sqcap Q$ is either a revivals of P or a revivals of Q.

- $$\begin{aligned} traces_M(P \,\Box\, Q) &= traces_M(P) \cup traces_M(Q) \\ deadlocks_M(P \,\Box\, Q) &= ((deadlocks_M(P) \cup deadlocks_M(Q)) \cap \{s \mid s \neq \langle\rangle\}) \\ &\quad \cup (deadlocks_M(P) \cap deadlocks_M(Q)) \\ revivals_M(P \,\Box\, Q) &= \{(\langle\rangle, X, a) \mid (\langle\rangle, X) \in failures^b_M(P) \cap failures^b_M(Q) \\ &\quad \wedge (\langle\rangle, X, a) \in revivals_M(P) \cup revivals_M(Q)\} \cup \\ &\quad \{(s, X, a) \mid (s, X, a) \in revivals_M(P) \cup revivals_M(Q) \wedge s \neq \langle\rangle\} \end{aligned}$$

$$\begin{array}{lcl} \mathit{failures}^b_M(P) & = & \{(s,X) \mid X \subseteq \Sigma \wedge s \in \mathit{deadlocks}_M(P)\} \cup \\ & & \{(s,X) \mid (s,X,a) \in \mathit{revivals}_M(P)\} \end{array}$$

$\mathit{failures}^b_M(P)$ records the failures of P from the stable states which *do not* terminate by signalling with the $\checkmark$ or engage in any internal events.

The process $P \;\Box\; Q$ will deadlock on the empty trace, when both the processes P and Q deadlock on the empty trace. When the trace is not the empty trace, only one of the subprocesses of P and Q is required to contribute.

Similarly, when the trace is empty, the refusal set of a revival of $P \;\Box\; Q$ must be from both $\mathit{failures}^b_M(P)$ and $\mathit{failures}^b_M(Q)$, and the reviving event of a revival comes from one of the revivals of P and Q. At the non-empty trace, only one of the processes of $P \;\Box\; Q$ is required to contribute to any revivals of $P \;\Box\; Q$.

- $$\begin{array}{lcl} \mathit{traces}_M(!!\ c : C \bullet P(c)) & = & \bigcup\{\mathit{traces}_M(P(c)) \mid c \in C\} \cup \{\langle\rangle\} \\ \mathit{deadlocks}_M(!!\ c : C \bullet P(c)) & = & \bigcup\{\mathit{deadlocks}_M(P(c)) \mid c \in C\} \\ \mathit{revivals}_M(!!\ c : C \bullet P(c)) & = & \bigcup\{\mathit{revivals}_M(P(c)) \mid c \in C\} \end{array}$$

  The deadlocks of $(!!\ c : C \bullet P(c))$ is the union of all deadlocks of $P(c)$ for each $c \in C$. Similarly the revivals of $(!!\ c : C \bullet P(c))$ is the union of all revivals of $P(c)$ for each $c \in C$.

- $$\begin{array}{lcl} \mathit{traces}_M(P \mathbin{|[}X\mathbin{]|} Q) & = & \{t_1 \mathbin{|[}X\mathbin{]|} t_2 \mid t_1 \in \mathit{traces}_M(P),\ t_2 \in \mathit{traces}_M(Q)\} \\ \mathit{deadlocks}_M(P \mathbin{|[}X\mathbin{]|} Q) & = & \{u \mid (s,Y) \in \mathit{failures}_M(P), (t,Z) \in \mathit{failures}_M(Q)\ . \\ & & Y - (X \cup \{\checkmark\}) = Z - (X \cup \{\checkmark\}) \\ & & \wedge\ u \in s \mathbin{|[}X\mathbin{]|} t \\ & & \wedge\ \Sigma^{\checkmark} = Y \cup Z\} \\ \mathit{revivals}_M(P \mathbin{|[}X\mathbin{]|} Q) & = & \{((u, Y \cup Z), a) \mid \\ & & \exists\, s,t.(s,Y) \in \mathit{failures}_M(P) \wedge (t,Z) \in \mathit{failures}_M(Q) \\ & & \wedge\ u \in s \mathbin{|[}X\mathbin{]|} t \cap \Sigma^* \\ & & \wedge\ Y - (X \cup \{\checkmark\}) = Z - (X \cup \{\checkmark\}) \\ & & \wedge\ ((a \in X \wedge (s,Y,a) \in \mathit{revivals}(P) \wedge \\ & & \qquad\qquad (t,Z,a) \in \mathit{revivals}_M(Q)) \\ & & \vee\ a \notin X \wedge (s,Y,a) \in \mathit{revivals}_M(P) \\ & & \vee\ a \notin X \wedge (t,Z,a) \in \mathit{revivals}_M(Q))))\ \}. \end{array}$$

$$\begin{array}{lcl} \mathit{failures}_M(P) & = & \{(s,X) \mid X \subseteq \Sigma^{\checkmark} \wedge s \in \mathit{deadlocks}_M(P)\} \cup \\ & & \{(s,X),(s,X \cup \{\checkmark\}) \mid (s,X,a) \in \mathit{revivals}_M(P)\} \cup \\ & & \{(s,X) \mid s ^\frown \langle\langle\rangle\rangle \in \mathit{traces}_M(P) \wedge X \subseteq \Sigma\} \cup \\ & & \{(s ^\frown \langle\checkmark\rangle, X) \mid s ^\frown \langle\langle\rangle\rangle \in \mathit{traces}_M(P) \wedge X \subseteq \Sigma^{\checkmark}\} \end{array}$$

Unlike $\mathit{failures}^b_M(P)$, $\mathit{failures}_M(P)$ records the failures of P from the stable states which *do not* engage in internal events only. This definition includes the event $\checkmark$ in the refusal sets like failures in the stable failures model. In Chapter 5, we discuss more about its properties. Calculating the deadlock component for the generalised parallel operator is not direct, as a deadlock can occur in a parallel network when none of the processes is deadlocked. It is calculated indirectly by first finding failures of individual processes. A deadlock s of $P \mathbin{|[}X\mathbin{]|} Q$ comes from the set $\{u \mid s \mathbin{|[}X\mathbin{]|} t\} \cap \Sigma^*$ where $(s,X) \in \mathit{failures}_M(P), (t,Z) \in \mathit{failures}_M(Q)$ and $X \cup Z$ should be equal to $\Sigma^{\checkmark}$. $X \setminus A^{\checkmark}$ and $Y \setminus A^{\checkmark}$ should be equal. The clause for the generalised parallel operator has become a little more complicated as we have to deal with both cases in which reviving event of the revival is, and is not in A. Any revivals will be combinations of its two processes.

- $$\begin{array}{lcl} traces_M(\texttt{IF } b \texttt{ THEN } P \texttt{ ELSE } Q) & = & \text{if } b \text{ evaluates to } True \text{ then } traces_M(P) \\ & & \text{else } traces_M(Q) \\ deadlocks_M(\texttt{IF } b \texttt{ THEN } P \texttt{ ELSE } Q) & = & \text{if } b \text{ evaluates to } True \text{ then } deadlocks_M(P) \\ & & \text{else } deadlocks_M(Q) \\ revivals_M(\texttt{IF } b \texttt{ THEN } P \texttt{ ELSE } Q) & = & \text{if } b \text{ evaluates to } True \text{ then } revivals_M(P) \\ & & \text{else } revivals_M(Q) \end{array}$$

If b evaluates to $True$, then the revivals of IF b THEN P ELSE Q are the revivals of P otherwise the revivals of IF b THEN P ELSE Q are the revivals of Q. Similarly, it is defined for the traces and deadlocks clauses.

- $$\begin{array}{lcl} traces_M(P \setminus X) & = & \{t \setminus X \mid t \in traces_M(P)\} \\ deadlocks_M(P \setminus X) & = & \{t \setminus X \mid t \in deadlocks_M(P)\} \\ revivals_M(P \setminus X) & = & \{(s \setminus X, Y, a) \mid (s, Y \cup X, a) \in revivals_M(P)\} \end{array}$$

The deadlocks of $P \setminus X$ are deadlocks of P such that events from X are removed from the deadlock traces. The revivals $P \setminus X$ is stable if the corresponding state of P refuses the whole of X. A revival (s, Y, a) is in $P \setminus X$ if P has a revival $(s, Y \cup X, a)$.

- $$\begin{array}{lcl} traces_M(P \rhd Q) & = & traces_M(P) \cup traces_M(Q) \\ deadlocks_M(P \rhd Q) & = & deadlocks_M(Q) \cup \{s \in deadlocks_M(P) \mid s \neq \langle\rangle\} \\ revivals_M(P \rhd Q) & = & revivals_M(Q) \cup \{(s, X, a) \in revivals_M(P) \mid s \neq \langle\rangle\} \end{array}$$

A deadlock of $(P \rhd Q)$ is either a deadlock of P such that the deadlock should not be equal to $\langle\rangle$ or a deadlock of Q. If P does not engage in any event initially, then it will eventually opt to engaging in as Q. Hence $\langle\rangle$ from $deadlocks(P)$ does not contribute to deadlocks of $(P \rhd Q)$. A revival (s, X, a) of $(P \rhd Q)$ is either a revival of P such that the trace s should not be equal to $\langle\rangle$ or a revival of Q.

- $$\begin{array}{lcl} traces_M(P[[R]]) & = & \{t \mid \exists\, t' \in traces_M(P).\ (t', t) \in [[R]]^*\} \\ deadlocks_M(P[[R]]) & = & \{t \mid \exists\, t' \in deadlocks_M(P).\ (t', t) \in [[R]]^*\} \\ revivals_M(P[[R]]) & = & \{(s', X, a') \mid \exists\, s, a\,.\, sR^*s' \wedge a\,R\,a' \\ & & \qquad \wedge\ (s, R^{-1}(X), a) \in revivals_M(P)\} \end{array}$$

The deadlocks of $P[[R]]$ are deadlocks of P such that each event is renamed according to the relation R. For each revival (s, X, a) of $P[[R]]$, the corresponding revivals of P has to refuse every single event which renames to the set X and should accept corresponding event that is renamed into the renamed reviving event.

- $$\begin{array}{lcl} traces_M(P \mathbin{;} Q) & = & (traces_M(P) \cap \Sigma^*) \cup \\ & & \{t_1 \frown t_2 \mid t_1 \frown \langle \checkmark \rangle \in traces_M(P),\ t_2 \in traces_M(Q)\} \\ deadlocks_M(P \mathbin{;} Q) & = & deadlocks_M(P) \cup \\ & & \{s \frown t \mid s \frown \langle \checkmark \rangle \in traces_M(P) \wedge t \in deadlocks_M(Q)\} \\ revivals_M(P \mathbin{;} Q) & = & \{(s, X, a) \mid (s, X, a) \in revivals_M(P)\} \cup \\ & & \{(s \frown t, X, a) \mid s \frown \langle \checkmark \rangle \in traces_M(P) \\ & & \qquad \wedge\ (t, X, a) \in revivals_M(Q)\} \end{array}$$

The deadlocks of $P \mathbin{;} Q$ has two clauses: the deadlocks of P before termination, and the terminating traces of P concatenation with deadlocks of Q. A stable revival of $P \mathbin{;} Q$ is either a revival of P or a terminating trace of P followed by a revival of Q.

- $$\begin{aligned} traces_M(P \bigtriangleup Q) &= traces_M(P) \cup \{s \frown t \mid s \in traces_M(P) \cap \Sigma^*, t \in traces_M(Q)\} \\ deadlocks_M(P \bigtriangleup Q) &= \{s \frown t \mid s \in traces_M(P) \cap \Sigma^* \wedge t \in deadlocks_M(Q)\} \\ revivals_M(P \bigtriangleup Q) &= \{(s, X, a) \in revivals_M(P) \mid (\langle\rangle, X) \in failures^b_M(Q)\} \\ &\cup\{(s, X, a) \mid (s, X) \in failures^b_M(P) \wedge (\langle\rangle, X, a) \in revivals_M(Q)\} \\ &\cup\{(s \frown t, X, a) \mid s \in traces_M(P) \cap \Sigma^* \wedge t \neq \langle\rangle \wedge (t, X, a) \in revival \end{aligned}$$

  The deadlocks of $P \bigtriangleup Q$ are the traces of P concatenated with deadlocks of Q such that the traces of P should not contain ✓. Only the deadlocks of Q contribute to deadlocks of $P \bigtriangleup Q$ as the control will be transferred to Q if P does not engage in any event. The revivals of $P \bigtriangleup Q$ are three clauses: revivals of P as long as $failures^b(Q)$ at the empty traces also refused the same refusal set of P; non terminating trace of P followed by a revival of Q such that revival trace should not be equal to ✓; if an initial event of Q engage at any moment after the trace s, then the revival after the trace s should contain refusal set of revivals of Q and of $failures^b(Q)$ as the refusal event and the initial event as reviving event.

- $$\begin{aligned} traces_M(P \lfloor n) &= \{t \in traces_M(P) \mid length(t) \leq n\} \\ deadlocks_M(P \lfloor n) &= \{d \in deadlocks_M(P) \mid length(d) \leq n\} \\ revivals_M(P \lfloor n) &= \{(t, X, a) \in revivals_M(P) \mid length(t) < n\} \end{aligned}$$

  The depth restriction operator $P \lfloor n$, which behaves exactly like P until exactly n events have occurred. Hence, the traces and deadlocks of $P \lfloor n$ are calculated by taking all the traces and deadlocks of length less than or equal to n respectively. For revivals clause, the length of the revival traces of $P \lfloor n$ should be strictly lesser than n.

- $$\begin{aligned} traces_M\ (\$p) &= fst(M(p)) \\ deadlocks_M\ (\$p) &= snd(M(p)) \\ revivals_M\ (\$p) &= thd(M(p)) \end{aligned}$$

  The meaning of process names (in CSP terms) is given by the environment M which is a function from process names into the domain of the stable revivals model. Hence the type of the semantic function is $CSP \longrightarrow Environment \longrightarrow domain(\mathcal{R})$. The semantic meaning of the process name for the deadlocks component is $deadlocks_M(\$p) = snd(M(p))$, where snd returns the second element of a triple. The semantic meaning of the process name for the revivals component is defined as $revivals_M(\$p) = thd(M(p))$, where thd returns the third element of a triple.

4.3 Responsiveness and Stuckness in the Stable Revivals Model

In Chapter 3, we focused on the definition of responsiveness and stuckness in the stable failures model of CSP and in CCS respectively. In this section, we give the definition of responsiveness and stuckness in the stable revivals model which captures these definitions precisely. We also see relations between the definitions of these properties in the stable revivals models and in the stable failures model.

The definition of RESPONDSTO in the model $\mathcal{R}$ is given in [Ros07] as

Definition 4.5: $\mathcal{R}-$RESPONDSTO. Let P, Q be processes. A Process Q $\mathcal{R}-$RESPONDSTO process P on J if there does not exist $(s, X, a) \in revivals_M(P)$ and $(t, Y) \in failures_M(Q)$ with $s \upharpoonright J =$

$t \upharpoonright J$ such that $a \in J$ and $(X \cap J) \cup Y = \Sigma^{\checkmark}$.

This says that whenever a process P wants co-operation from process Q an event $a \in J$, then the process Q must not refuse it. We reproduce the following lemma given in [RRS06].

Lemma 4.6: Q $\mathcal{R}-$RESPONDSTO P implies Q RESPONDSTOLIVE P.

Proof. If parallel process $P\|[J]\|Q$ block each other on $J^{\checkmark}$, then by definition of parallel composition, the failures must have been created by its sub-processes. If $(s, J^{\checkmark}) \in failures_M(P\|[J]\|Q)$ is created by maximum failures of $(s \upharpoonright \alpha P, X)$ of P and $(s \upharpoonright \alpha Q, Y)$ of Q. But the definition of $failures_M(P)$, $(s \upharpoonright \alpha P, X)$ of P comes either from a deadlock trace $s \upharpoonright \alpha P$ of P or a revival $(s \upharpoonright \alpha P, X, b)$ of P with $b \notin J$. In the second case, by definition of healthiness condition of $R2$, and $\mathcal{R}-$RESPONDSTO, we get $J \subseteq X$. Hence in either cases we have, $(s \upharpoonright \alpha P, J^{\checkmark}) \in failures_M(P)$. □

Q $\mathcal{R}-$RESPONDSTO P says that reviving events happen in the same stable state whereas the old definition in the stable failures model says that reviving events may not necessarily happen in the same stable state [Ros07]. By this argument, it is clear that Q RESPONDSTO P implies Q $\mathcal{R}-$RESPONDSTO P.

Definition 4.7: Deterministic process. A process is deterministic if it is divergence free, and after any trace, cannot both accept and refuse the same event at the same time.

If Q is deterministic, then all the three conditions are same. Hence we can use of one of the definitions which are easier to check in the existing tool. The concept of ✓ in CSP gives a solution of stuckness as deadlock. The following definition gives the CCS style definition of stuckness in CSP [Ros07].

Definition 4.8: $\mathcal{R}$-stuck-free. A process N is $\mathcal{R}$-stuck-free with respect to a set of actions A provided it has no revivals of the form $(s, \Sigma - A, a)$ with $s \in (\Sigma - A)^*$ and $a \in A$.

In CCS, $P \mid Q$ allows the processes to perform events asynchronous or to perform events synchronised in which case they are hidden and become τ. This makes stuckness easier to define in CCS. In order to capture this notion of CCS in CSP, we rename all the processes in a network so that every synchronised event is mapped to both itself and a new event called *stuck* that is not synchronised. The renamed network is struck-free if it does not have the revival $(s, \Sigma - \{stuck\}, stuck)$ for any $s \in (\Sigma - \{stuck\})^*$ [RRS06].

Consider for example the process

$$a \rightarrow STOP \;\|[\{a, b\}]\|\; b \rightarrow STOP$$

The process is stuck as we have $(\langle\rangle, \{a, b\}, stuck)$ in the renamed process

$$a \rightarrow STOP \,\Box\, stuck \rightarrow STOP \;\|[\{a, b\}]\|\; b \rightarrow STOP \,\Box\, stuck \rightarrow STOP$$

In CCS, this process can be written as $(a.\mathbf{0} \mid b.\mathbf{0}) \setminus \{a, b\}$.

4.4 Relationship between the Stable Revivals Model and other models

In this section, we discuss the stable revivals models in the hierarchy of CSP models. The models which come closest with the stable revivals model are

- The *stable failures model* $\mathcal{F}$ captures a process P in terms of (T, F) where T is same as in the first component of the stable revivals model and F is a set of pairs (s, X) with s a trace and X a refusal set. If a process P is represented as (T, D, R) in the stable revivals model, we can get the F value of P from (T, D, R) by the calculation given in Section 4.3, hence the stable revivals model is finer than the stable revivals model. The distinction between failures and revivals is that revivals guarantee acceptance of an event at a $\checkmark$-stable state after trace s. The stable failures model is useful to capture deadlock situations of a process.
- The *stable ready sets model* $\mathcal{A}$ in which a process is represented by (T, R) where T is the same as in the first component of the stable revivals model and R is a set of ready sets (s, X). A ready set X is a set of events that the process is ready to engage in. In a Labelled Transition System (LTS), a ready set can be calculated by taking union of all labels on the outgoing events from a stable node. The model $\mathcal{A}$ is more discriminative than the models $\mathcal{F}$ and $\mathcal{R}$. Consider for example the processes
$$P = a \longrightarrow \mathtt{STOP} \sqcap b \longrightarrow \mathtt{STOP}$$
$$Q = a \longrightarrow \mathtt{STOP} \sqcap b \longrightarrow \mathtt{STOP} \sqcap (a \longrightarrow \mathtt{STOP} \,\Box\, b \longrightarrow \mathtt{STOP})$$
The ready sets of P are $\{(\langle\rangle, \{a\}), (\langle\rangle, \{b\})\}$, and for Q, they are $\{(\langle\rangle, \{a\}), (\langle\rangle, \{b\}), (\langle\rangle, \{a, b\})\}$. The failures of P and Q are identical, i.e., $\{(\langle\rangle, \{a\}), (\langle\rangle, \{b\})\}$. Similarly the revivals of P and Q are also identical, i.e., $\{(\langle\rangle, \{a\}, b), (\langle\rangle, \{b\}, a)\}$.
- The *refusal Testing model* $\mathcal{RT}$ is based on refusal testing in CCS. In this model, a process is identified by a finite alternating sequence of the form
$$\langle X_0, a_0, X_1, a_1, \ldots, a_n, X_{n+1}\rangle$$
where each a_i is a visible event and each X_i is either a $\checkmark$-stable refusal observable at the appropriate time or a marker $\bullet$ indicating no refusal has been observed.

In the rest of the section, we describe the relations between the above model for the core language of CSP in terms of *congruence*. We closely follow the definitions as described in [Ros07]. We present only the results. Congruence is a notion of equivalence for processes that is compositional under all operators of a language. Each semantic model of CSP induces a congruence for CSP. In a congruence, no context can map two equivalent processes to two in-equivalent ones [Ros07].

If **S** is a nonempty collection of congruences, define a sub-**S** congruence to be one that does not distinguish any pair of processes equated by any member of **S**. If **S** is a singleton, we write sub-S. If χ is sub-Υ, we write $\chi \preceq \Upsilon$. The following important lemmas are proven in [Ros07].

Lemma 4.9: If $\mathcal{M}$ is any sub-$\{\mathcal{RT}, \mathcal{A}\}$ congruence then $\mathcal{M} \preceq \{\mathcal{RT}, \mathcal{A}\}$ satisfies $\mathcal{M} \preceq \mathcal{R}$.

It has also been proven that in the below sequence, any sub-$\mathcal{R}$ congruence that is strictly less abstract than any non-final member of the sequence.

$$\mathcal{NULL}, \quad \mathcal{T}, \quad \mathcal{F}, \quad \mathcal{R}$$

where $\mathcal{NULL}$ identifies all processes. The above result establishes the important result about the hierarchy of CSP.

Theorem 4.10: For the core CSP, the only sub-$\{\mathcal{RT}, \mathcal{A}\}$ congruences are $\mathcal{T}$, $\mathcal{F}$, $\mathcal{R}$ and $\mathcal{NULL}$.

In [Ros07], it is proven that the traces model, the stable failures model and the stable revivals model are successively more refined. Thus two processes which are equivalent in $\mathcal{R}$ are also equivalent in $\mathcal{F}$. It is also proven that there does not exist any model that refines the stable revivals model and more abstract the stable acceptance model and the refusal-testing model of CSP. Hence, it has an important place in the hierarchy of CSP models.

Chapter 5

Implementation of the Stable Revivals Model

Contents

In this chapter, we discuss the implementation of the stable revivals model. Firstly, we present the architecture of CSP-Prover with the new model. Secondly, we discuss the implementation of the domain of the model and finally, we discuss the encoding of the semantic functions of the model in the CSP-Prover.

5.1 Architecture of the CSP-Prover

CSP-Prover is developed to have a generic architecture. CSP-Prover is designed to make it suitable to plug in various denotational semantics of CSP. It is envisioned that any new CSP model can be easily added without any major difficulty. Figure 5.1 shows the architecture of CSP-Prover with the stable revivals model. The reusable part contains Tarksi fixed point theorem and the standard fixed point induction rule based on complete partial orders (CPO). CPO and pointed CPO are defined as axiomatic type classes. It also has Banach's fixed point theorem and the metric fixed point induction rule based on complete metric spaces (CMS).

For the stable revivals model, we implement the theory based on Complete Partial Orders (CPO) to assign the meaning to recursive processes. We prove the domain of the model $\mathcal{R}$ is a CPO by making the domain of the model $\mathcal{R}$ as an instance of pointed CPO, an axiomatic classes which is defined in the reusable part of CSP-Prover. The models $\mathcal{T}$ and $\mathcal{F}$ support both theories of CMS and CPO.

The instantiated part for each model consists of theories for the domain of the model, the semantic functions of the model and the proof infrastructure. Hence the encoding of a model involves the three major parts. We briefly explain the elements of the instantiated part of $\mathcal{T}$ as we will be using lemmas in this model frequently and we also follow the same strategy as used in this model. The type of domain of the traces model $\mathcal{T}$ is `'a domT`.

```
consts
 HC_T1 ::  "'a trace set => bool"
defs
 HC_T1_def :  "HC_T1 T == (T~= {} & prefix_closed T)"
typedef 'a domT = "{T::('a trace set).  HC_T1(T)}"
apply (rule_tac x ="{⟨⟩}" in exI)
by (simp add:  HC_T1_def prefix_closed_def)
```

The above code creates the type of domain `'a domT` where $'a$ is a type variable. The predicate `HC_T1_def` checks the healthiness condition of the model $\mathcal{T}$ i.e., whether a set of traces is non-empty or not and whether it is prefixed closed or not. This is defined in theory called `Domain_T`. `Domain_T` also contains lots of useful theorems such as the domain of the model $\mathcal{T}$ is closed under union.

The semantic function for the model $\mathcal{T}$ is defined in theory `CSP_T_semantics`.

```
theory CSP_T_semantics
imports CSP_syntax Domain_T_cms
          Trace_par Trace_hide Trace_ren Trace_seq
```

The theory `CSP_T_semantics` uses the theory of CSP syntax, the theories of the parallel operator, the hiding operator, the renaming operator, and the sequential operator from the reusable part of CSP-Prover. It also uses theory on CMS through theory `Domain_T_CMS` in the above code. Hence it provides the option to the user to select CPO theory or CMS theory. CMS theory guarantees uniqueness property for guarded processes, but CPO theory works for any processes without the uniqueness property. The uniqueness property guarantees the unique solution to the recursive definition.

The type of semantic function of the model $\mathcal{T}$ is declared as below:

```
consts
 traces ::  "('p,'a) proc => ('p => 'a domT) => 'a domT"
```

The semantic function is defined by primitive recursion over the process using the keyword `primrec` as follows: type `('p,'a) proc`.

```
primrec
" traces(STOP) = (%M. {⟨⟩}t)"
" traces(SKIP) = (%M. { ⟨⟩, ⟨ Tick ⟩ }t)"
```

The CSP traces semantic function is defined by translating processes into the domain of the model $\mathcal{T}$ using a constant function `semTf` defined as follows:

```
consts
 semTf :: "('p,'a) proc => ('p => 'a domT) => 'a domT"  ("[[_]]Tf")
 semTfun ::  "('p => ('p,'a) proc) => ('p => 'a domT) => ('p => 'a
domT)"  ("[[_]]Tfun")
defs
 semTf_def: "[[P]]Tf == (%M. traces(P) M)"
 semTfun_def: "[[Pf]]Tfun == (%M. %p.  [[Pf p]]Tf M)"
```

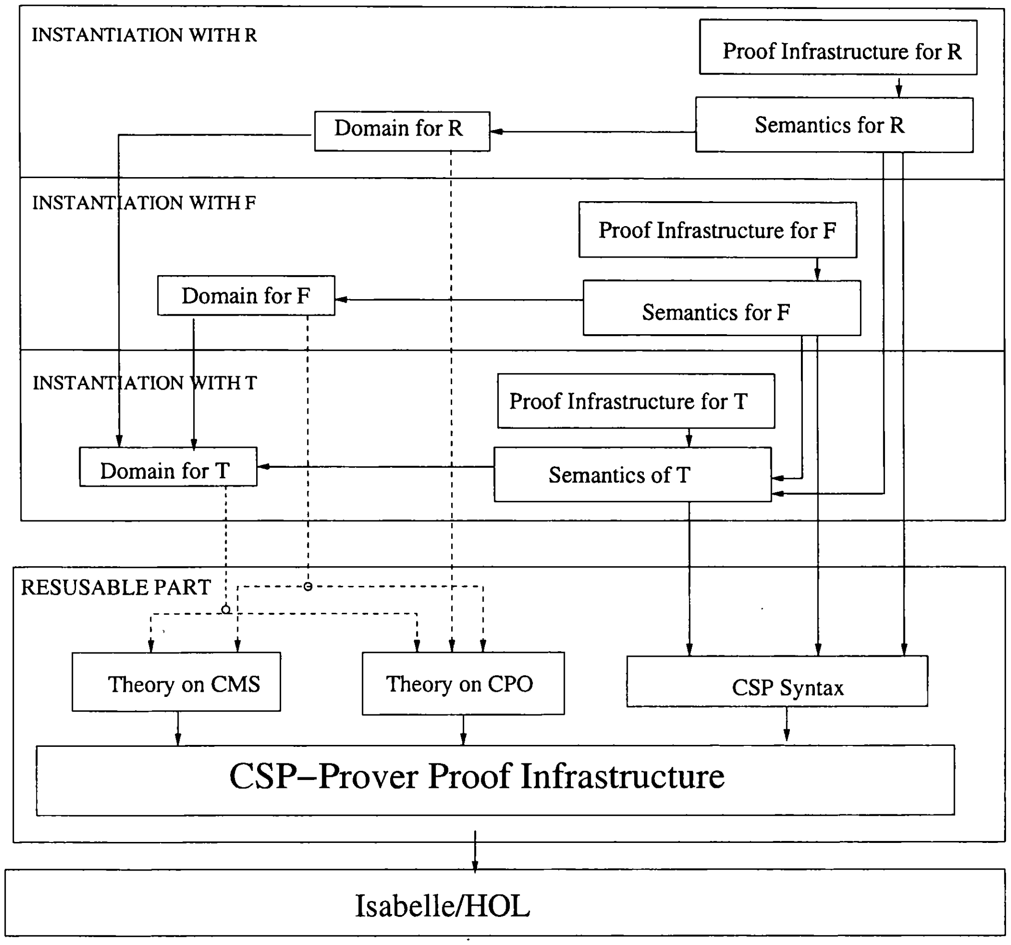


Figure 5.1: Architecture of the model R in CSP-Prover

The definition of semTfun binds process names to processes. The meaning of each recursive function is defined by semTfix. Finally the proof infrastructure of each model contains the step laws for operators, basic laws, distributive laws, fixed-point induction rules, and other laws which required to convert process terms into normal form. It also contains tactics that are required to simplify complex processes.

We implemented the domain of the stable revivals model using the implementation of the trace model in CSP-Prover. Similarly in the implementation of the semantic function of the model, we used some theories from the implementation of the stable failures model in CSP-Prover. Though we could have implemented it from scratch, but it would be reinventing the wheel (developing the code of around 15K lines) as one of the component in the stable revivals model has the same property as the domain of the traces model. It also shows the versatile property of CSP-Prover implementation and establishes a close link between the models of CSP. We explain more about the exact theories that we reuse in the encoding of the domain and the semantics function of the model $\mathcal{R}$ in the next sections.

5.2 Implementation of the domain of the model $\mathcal{R}$

We implement the domain of the model $\mathcal{R}$ on the top of the domain of the model $\mathcal{T}$. The domain of the model $\mathcal{T}$ is implemented in theory *Domain_T*. We import the theory of the domain of the traces model in CSP-Prover using the following command:

```
theory Domain_R
imports Set_D Set_R Domain_T
```

The above command also imports two more theories, namely, two theories that implement for the deadlocks and the revivals component. The hierarchy of theory files of the implementation is given in Figure 5.2.

5.2.1 Implementation of the deadlock component

The implementation of the deadlock component is similar to the implementation of the domain of the model $\mathcal{T}$. Since a deadlock in the deadlock component of the model $\mathcal{R}$ is same as a trace except that events in deadlocks do not have $\checkmark$ at the end, we define a predicate to check this. The predicate HC_DO T checks whether the traces in the set of traces T contains *Tick* or not. The below code shows the implementation of HC_DO T.

```
consts
 HC_D0 ::  "'a trace set => bool"
defs
 HC_D0_def :  "HC_D0 T == (ALL s .  s :  T --> Tick ~:  sett
(s))"
```

We create a new type to represent the deadlock component using the following code:

```
typedef 'a setD = "{F::('a trace set ).  HC_D0(F) }"
apply (rule_tac x ="{⟨⟩}" in exI)
apply (simp add:  HC_D0_def )
done
```

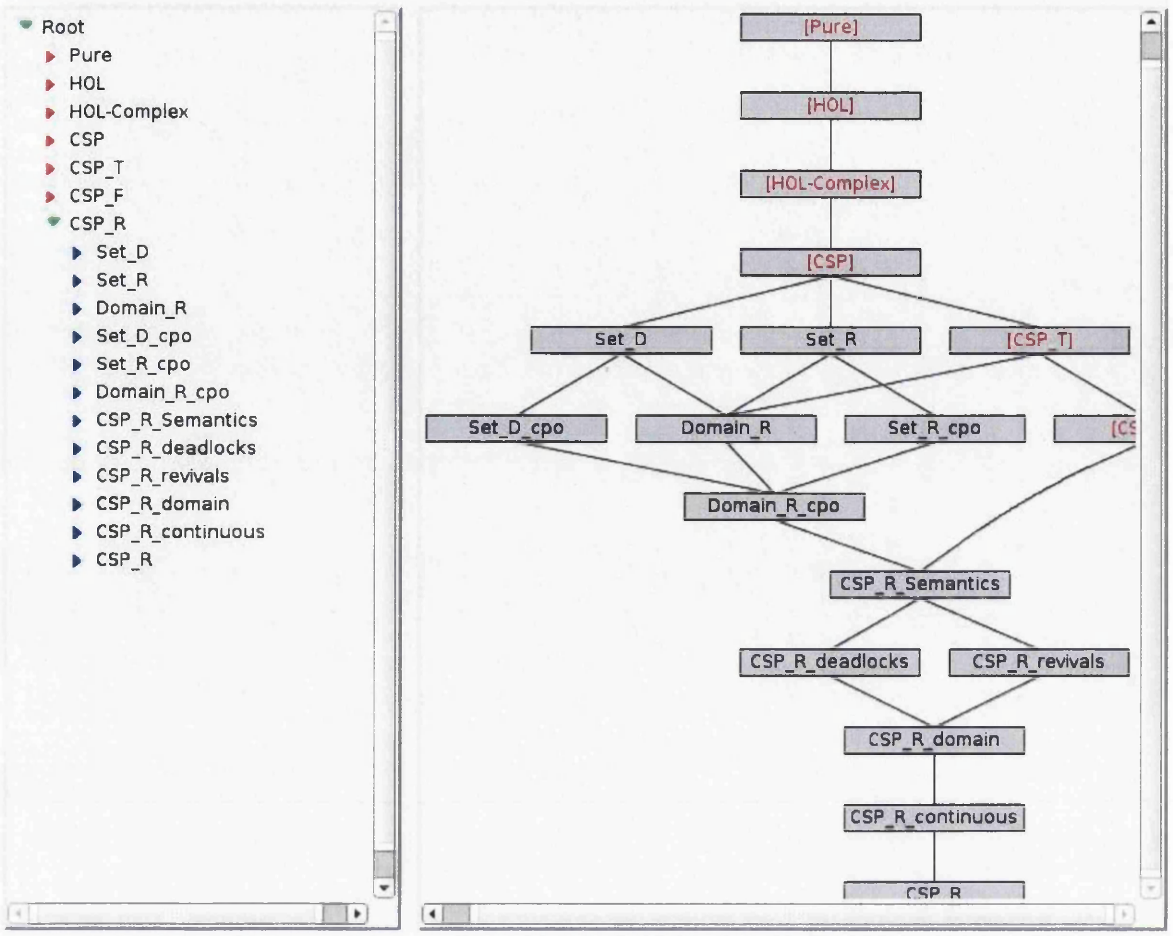


Figure 5.2: The domain and semantics encoding of the stable revivals model

The above code creates a set and a type with the same name `'a setD` which contains all the traces set that satisfies the predicate `HC_DO`.

We could have implemented the deadlock component without using the definition of trace `'a trace` by using `'a list`, as values in the type definition of `'a list set` and `'a setD` are the same. However we created `'a setD` from `'a trace set` for the following reasons:

- We frequently need to compare deadlocks from the deadlock component with traces from the traces component in the stable revivals model. If we had created the deadlock component as

  ```
  types 'a setD = 'a list set
  ```

  then we would not have been able to compare `'a trace` with `'a list` without any modification. Even if we had written some comparison functions which do the above job, then in most of cases it is inevitable that we must prove most of theorems without condition similar to the predicate `HC_DO`.

- Building up the deadlock component from scratch may need another 10000 lines of codes to define theories for the parallel operator, the renaming operator, the hiding in terms of `'a list`, as all the theories have been implemented in terms of `'a trace` in CSP-Prover.

The set `'a setD` is a subset of the set `'a trace set set`. Isabelle creates the following three constants to convert to and fro between the abstract and deadlock representations:

```
setD ::  'a trace set set
Rep_setD ::  'a setD => 'a trace set
Abs_setD ::  'a trace set x=> 'a setD
```

It also asserts that `Rep_setD` is surjective on the subset of type `'a trace set` and asserts that `Rep_setD` and `Abs_setD` are inverse of each other. These assertions are proven by the following lemmas: `Rep_setD`, `Rep_setD_inverse`, and `Abs_setD_inverse`.

Isabelle creates other useful lemmas like injectivity of `Rep_setD` and `Abs_setD`. By declaring `Rep_setD T` using the attribute `simp`, we allow Isabelle to apply simplification of `Rep_setD T : setR` automatically as follows:

```
declare Rep_setD [simp]
```

This simplification applies globally whenever we apply command `apply simp`. We define the following constant definitions:

```
consts
 memD:: "'a trace => 'a setD => bool" ("(_/ :d _)" [50, 51] 50)
 CollectD ::  "('a trace => bool) => 'a setD"("CollectD")
 UnionD ::  "'a setD set => 'a setD" ("UnionD _" [90] 90)
 InterD ::  "'a setD set => 'a setD" ("InterD _" [90] 90)
 empD ::  "'a setD" ("{}d")
 UNIVD ::  "'a setD" ("UNIVd")
defs
 memD_def :  "x :d D == x :  (Rep_setD D)"
 CollectD_def :  "CollectD P == Abs_setD (Collect P)"
 UnionD_def :  "UnionD Ts == Abs_setD (Union (Rep_setD ` Ts))"
 InterD_def :  "InterD Ts == Abs_setD (Inter (Rep_setD ` Ts))"
 empD_def :  "{}d == Abs_setD {}"
 UNIVD_def :  "UNIVd == Abs_setD UNIV"
```

`t :d D` is syntactic sugar for `memD t D`. `memD t D` checks membership of an element t in the set D which is of type `'a setD` i.e., whether trace t is an element of the deadlock set `Rep_setD D`.

`UnionD Ts` defines the union of sets `Ts`. Each element in `Ts` is of type `'a setD`. We calculate `UnionD Ts` by first converting each element in `Ts` into `'a trace set` representation using `Rep_setD ` Ts` and then perform the union of those using the `'a trace set` union operator `Union` and then convert it back to its abstract representation using `Abs_setD`. Similarly `InterD Ts` performs the intersection of the set `Ts` which is of type `'a setD`.

`empD` returns the empty set in the type `'a setD`. The empty set in `'a setD` corresponds to the empty set in `'a trace set`. Similarly the universal set in `'a setD` corresponds to the universal set in `'a trace set`. We declare the type `'a setD` to be an instance of Isabelle axiomatic type class `ord` using the following commands:

```
instance setD ::  (type) ord
by (intro_classes)
```

We overload the definition of < and <= in terms of `'a setD`. This is done by converting `'a setD` representation into `'a trace set` represent

```
defs (overloaded)
 subsetD_def: "T <= S == (Rep_setD T) <= (Rep_setD S)"
 psubsetD_def: "T < S == (Rep_setD T) < (Rep_setD S)"
```

Similarly we declare the type `'a setD` to be an instance of axiomatic type class `order`. Here we need to prove that it satisfies the following axioms of the partial order definitions defined in theory `Orderings` in HOL.

```
 order_refl [iff]:  "x <= x"
 order_trans:  "x <= y ==> y <= z ==> x <= z"
 order_antisym:  "x <= y ==> y <= x ==> x = y"
 order_less_le:  "(x < y) = (x <= y & x ~= y)"
```

Now we prove the following useful general lemmas:

```
lemma setD_D0:  "[| T : setD ; s :  T |] ==> Tick ~:  sett s"
lemma memD_D01:  "s :d T ==> noTick s"
```

The first lemma states that if T is in set `setD` and trace s is in T, then ✓ does not appear in the elements of the trace t. It shows that all the traces in T does not contain ✓. The second lemma also states the same but here it says that if T is of type `setD` and s is in `Rep_domT T`, then ✓ does not appear in trace s.

Then we prove that the union of `(Rep_setD ` Fs)` is in `setD` by the following lemma:

```
lemma setD_Union_in_setD: "(Union (Rep_setD ` Fs)) :  setD"
```

where `Fs` is of type `'a setD set`. This states that the elements in the `setD` are closed under the union operation. This will be useful when proving the set `setD` is a CPO.

5.2.1.1 set_D is a pointed cpo

Theory `set_D_cpo` implements the proofs that `set_D` is a pointed complete partial order (CPO). It imports two theories `set_D` and `CPO` using the below command:

```
theory Set_D_cpo = Set_D + CPO :
```

Recall that in CSP-Prover, to prove that `set_D` is a pointed cpo we need to prove that `set_D` is an instance of axiomatic type class `cpo_bot`. To prove `set_D` is an instance of axiomatic type class `cpo_bot`, we need to prove that `set_D` is an instance of axiomatic type class `cpo` and `bot`. Hence we show that it satisfies the following axioms of `cpo` and `bot` respectively.

```
bottom_bot :  "Bot <= (x::'a::bot0)''
complete_cpo :  "(directed (X::'a::order set)) ==> X hasLUB"
```

We declare that `set_D` is an instance of `bot0` and then overload the `Bot` element with the empty set of *setD*.

```
instance setD ::  (type) bot0
by (intro_classes)
defs (overloaded)
 bottom_setD_def :   "Bot == {}d"
```

We prove the following important lemmas to satisfy the axiom `complete_cpo`.

```
lemma UnionD_isUB : "(UnionD Fs) isUB Fs"
lemma UnionD_isLUB : "UnionD Fs isLUB Fs"
```

The above lemmas establishe that `UnionD Fs` is the upper bound and is the least upper bound of set `Fs`. Finally we declare that `set_D` to be an instance of `cpo` and `cpo_bot`.

```
instance setD ::  (type) cpo
instance  setD ::  (type) cpo_bot
```

5.2.2 Implementation of the Revivals component

The third component in the stable revivals model is implemented in theory file `set_R.thy`. Organisation and proof methods are very similar to the deadlock component, however it is little bit complex as each revivals in the revivals component has three elements. It is implemented using the theory `Traces`. Recall that the signature of the revivals component is $(\Sigma^* \times \mathcal{P}(\ \Sigma) \times \Sigma)$. We create a type synonym called `'a revival` to represent a revival in the revivals component without any restriction.

```
types 'a revival = "('a trace * 'a event set * 'a event) "
consts
 FstR ::  "'a revival => 'a trace "
 SndR ::  "'a revival => 'a event set "
 ThdR ::  "'a revival => 'a event "
defs
 FstR_def :   "FstR == (%F. fst(F))"
 SndR_def :   "SndR == (%F. fst(snd(F)))"
 ThdR_def :   "ThdR == (%F. snd (snd(F)))"
```

Isabelle internally represent the triple (s, X, a) as $(s, (X, a))$. We use projection functions from the product constructor $(*)$ to extract individual elements from a triple. The constant functions `FstR`, `SndR` and `ThdR` extract the first, second and third elements of a revival respectively. If $(s, X, a) \in R$, then the trace s has no $\checkmark$, the refusal set X does not contain $\checkmark$, and the event a is not equal to $\checkmark$. We encode this condition with the following below constant predicates:

```
consts
 HC_RT ::  "'a revival set => bool"
 HC_RF ::  "'a revival set => bool"
defs
 HC_RT_def :   "HC_RT F == (ALL f .( f :  F)
                                   --> Tick ~:  sett(FstR(f)))"
 HC_RF_def :   "HC_RF F == (ALL f .( f :  F) --> Tick ~:  SndR(f)
&
               snd(snd(f)) ~:  fst(snd(f)) & snd(snd(f))~=Tick )"
```

The constant predicate HC_RF also includes the condition HC_R3 which states that $a \notin X$ for a revival (s, X, a). We also encode the healthiness conditions HC_R2 and HC_R3 in the revivals component as these conditions are related to the revivals component only. The healthiness conditions are given below:

```
consts
 HC_R2 ::  " 'a revival set => bool"
 HC_R3 ::  " 'a revival set => bool"
defs
 HC_R2_def :   "HC_R2 F == ALL s X a Y. ( (s,X,a) :  F & Y <= X &
                Tick  :  X & noTick s --> (s,Y,a) :  F )"
 HC_R3_def :   "HC_R3 F == ALL s X b c.  ((s,X,b) :  F &
                Tick ~:  X & noTick s & c~=Tick -->
               ((s,X,c)  :  F | (s,X Un { c } ,b)  :  F ) )"
```

Below we summarise the conditions included in the revivals component:

- HC_RT: $\forall\ s X\ a\,.(s, X, a) \in R \Longrightarrow \checkmark \notin sett(s)$
- HC_RF: $\forall\ s X\ a\,.(s, X, a) \in R \Longrightarrow \checkmark \notin X \wedge \checkmark \neq a \wedge a \notin X$
- HC_R2: $\forall\ s X\ a\ Y\,.(s, X, a) \in R \wedge Y \subseteq X \Longrightarrow (s, Y, a) \in R.$
- HC_R3: $\forall\ s X\ a\ b\,.(s, X, a) \in R \wedge b \in \Sigma \Longrightarrow ((s, X, a) \in R \vee (s, X \cup \{b\}, a) \in R).$

HC_RF encodes the condition **RRS05**. In the below code, we define the type of the revivals component in the model $\mathcal{R}$ by encoding the above conditions:

```
typedef 'a setR = "{ R ::('a revival set ) .  HC_RT(R) & HC_RF(R) &
                                     HC_R2(R) & HC_R3(R)}"
apply (rule_tac x ="{}" in exI)
by (simp add:  HC_RT_def HC_RF_def HC_R2_def HC_R3_def)
```

We finish the proof by giving the witness {} to show that the new type is non-empty.

Similar to the type set_D, we create constant functions to check membership, to calculate the union of elements in the set set_R, to return the empty set and the universional set in the type set_R. We declare the type set_R to be an instance of axiomatic type class ord and order by the following commands:

```
instance setR ::  (type) ord
by (intro_classes)
defs (overloaded)
 subsetR_def  :   "F <= E == Rep_setR (F) <= Rep_setR (E)"
 psubsetR_def  :   "F < E == Rep_setR (F) < Rep_setR (E)"
instance setR ::  (type) order
```

We overload the definitions of < and <= in terms of `'a setR`. To declare `set_R` is to be an instance of `ord`, and we prove that the set `set_R` is a partial order. A lot of useful lemmas have been proven in this theory file. Some of them are given below:

```
lemma memR_RT: "r :r R ==> Tick ~: sett (FstR(r))"
lemma setR_RT: "[| R : setR ; r : R |] ==> Tick ~: sett(FstR(r))"
lemma setR_RF: "[| R : setR ; r : R |] ==> Tick ~: SndR(r) &
snd(snd(r)) ~: fst(snd(r)) & snd(snd(r)) ~=Tick"
lemma memR_RF: "r :r R ==> Tick ~: SndR(r) & snd(snd(r)) ~:
fst(snd(r)) & (snd(snd(r))~=Tick
lemma setR_R2: "[| R : setR ; (s,X,a) : R ; Y <= X ; Tick ~: X
; noTick s |] ==> (s,Y,a) : R"
lemma memR_R2: "[| (s,X,a) :r R ; Y <= X |] ==> (s,Y,a) :r R"
lemma setR_R3: "[| R : setR ; (s,X,b) : R ; Tick ~:X ; noTick s
; c~=Tick |] ==> (s,X,c) : R | (s,insert c X ,b) : R"
lemma memR_R3: "[| (s,X,b) :r R ; c~=Tick |] ==> (s,X,c) :r R |
(s,insert c X ,b) :r R"
```

All the lemma whose name starts with `setR_`, indicates that if R is in set `setR`, then the respective healthiness conditions hold. All the lemmas which names start with memR_ , states that if R is of type `setR`, then healthiness conditions hold. For example, in the lemma `setR_R2` if R is in set `setR`, $(s, X, a) \in R$, $Y \subseteq X$, then $(s, Y, a) \in R$. In the lemma memR_R2, if R is of type `setR`, $(s, X, a) \in Rep\_setR\ R$, and $Y \subseteq X$, then $(s, Y, a) \in Rep\_setR\ R$.

We prove that `setR` is closed under union as stated in the following:

```
lemma setR_Union_in_setR: "(Union (Rep_setR ` Fs)) : setR"
```

This will be useful in proving `setR` is a CPO. That `setR` is a pointed CPO is proved in theory `setR_cpo`. The theory `setR_cpo` is extended as a successor of the theory `setR`. `setR_cpo` declares that `setR` is an instance of `cpo` and `cpo_bot`. To prove it, we prove the following lemmas:

- UnionR Rs is an upper bound of set Rs:

```
lemma UnionR_isUB : "(UnionR Fs) isUB Fs"
```

- UnionR Rs is the least upper bound of set Rs:

```
lemma UnionR_isLUB : "UnionR Fs isLUB Fs"
```

- The least upper bound of Fs is UnionR Rs:

```
lemma isLUB_UnionR_only_if: "F isLUB Fs ==> F = UnionR Fs"
```

The below code proves that setR is a CPO and a pointed CPO:

```
instance setR ::  (type) cpo
apply (intro_classes)
...
instance setR ::  (type) cpo_bot
by (intro_classes)
```

5.2.3 Encoding the domain of the model $\mathcal{R}$

The domain of the model $\mathcal{R}$ is implemented in the theory Domain_R. The theory Domain_R is based on the theories Domain_T, set_R, set_D. The imported theories implement the individual components of the model $\mathcal{R}$.

To increase readability of the code, we create a type synonym called 'a domTsetDsetR which denotes a triple representing the domain of the trace component, the domain of the deadlock component and the domain of the revivals component. We also declare projection functions on the components of the triple.

```
types 'a domTsetDsetR = " ( 'a domT * 'a setD * 'a setR)"
consts
 Fst ::   "'a domTsetDsetR => 'a domT "
 Snd ::   "'a domTsetDsetR => 'a setD "
 Thd ::   "'a domTsetDsetR => 'a setR "
defs
 Fst_def  :   "Fst == (%F. fst(F))"
 Snd_def  :   "Snd == (%F. fst(snd(F)))"
 Thd_def  :   "Thd == (%F. snd(snd (F)))"
```

We select the individual elements in the triple 'a domTsetDsetR by using projection functions fst and snd. The constant functions Fst, Snd, and Thd return elements in the domain of traces component, the domain of deadlocks component and the domain of revivals component respectively.

We know that in the traces component, we implemented the healthiness condition HC_T1 and in the revivals component, we implemented the healthiness condition HC_R2, HC_R3 and HC_RRS. As we have already implemented the healthiness conditions HC_T1 , HC_R2, HC_R3, and RRS. Now we encode only the following two conditions in the domain which relate component with each other, namely HC_D1 deadlocks component with the traces component and HC_D1 revivals component with the traces component:

- $HC\_D1 : D \subseteq T$.
- HC_R1: $(s, X, a) \in R \Rightarrow s \frown \langle a \rangle \in T$.

The below code defines the constant definitions:

```
consts
 HC_D1 ::  " 'a domTsetDsetR => bool "
 HC_R1 ::  " 'a domTsetDsetR => bool "
defs
 HC_D1_def :
" HC_D1 T == (Rep_setD (Snd(T))) <= (Rep_domT (Fst(T)))"
 HC_R1_def :  " HC_R1 T == ALL s X a.  ( (s,X,a) :r Thd(T)
                             --> s ^^ <a> :t Fst(T))"
```

The constant definition `tripleR` combines the individual components to form an element in the domain of the model $\mathcal{R}$. It is defined as below:

```
consts
 tripleR ::  "'a domT => 'a setD => 'a setR => 'a domR"
defs
 tripleR_def :  "tripleR == %T D R. (Abs_domR (T,D,R))"
```

We define the following constant definitions to select the individual components in the domain.

```
consts
 fstR ::  "'a domR => 'a domT"
 sndR ::  "'a domR => 'a setD"
 thdR ::  "'a domR => 'a setR"
defs
 fstR_def :  "fstR == Fst o Rep_domR"
 sndR_def :  "sndR == Snd o Rep_domR"
 thdR_def :  "thdR == Thd o Rep_domR"
```

We prove lemmas concerning the healthiness conditions for elements in the domain.

```
lemma FstR_SndR_ThdR_in_domR[simp]:
              "(fstR SF , sndR SF, thdR SF) : domR"
lemma FstR_SndR_ThdR_domR[simp]:
              "(tripleR (fstR SF) (sndR SF) (thdR SF) ) = SF"
```

We prove an important lemma to decompose the semantic function of the stable revivals model into the individual components. This directly follows from injectivity of the domain of the stable revivals model and the definition of pair-wise decompose.

```
lemma eqR_decompo: "(SF = SE) =
(fstR SF = fstR SE & sndR SF = sndR SE & thdR SF = thdR SE)"
```

We finish the theory by proving monotonic property of individual component.

5.3 Encoding the semantic functions

In the previous section, we have shown how the domain of the stable revivals model has been implemented in CSP-Prover. In this section we will explain the encoding of the semantic functions of the model in CSP-Prover. Our approach to the encoding is slightly different from the implementation of other models implemented in CSP-Prover. First we explain the reason why we are not able to directly implement it as the stable failures model in CSP-Prover. Then we explain the encoding by a method we called 'two-phase construction'.

5.3.1 Problem in the encoding

In the stable failures model, the semantic functions of *traces*(P) and *failures*(P) are defined separately. This paved the way for reusing the semantic function *traces*(P) from the traces model. This also helps us to prove properties like the type correctness, continuity, etc easily. It also makes the code more readable.

We could have done the same for the stable revivals model by defining the semantic function separately as below:

```
consts
 deadlocks ::  "('p,'a) proc => ('p => 'a domR) => 'a setD"
 revivals ::  "('p,'a) proc => ('p => 'a domR) => 'a setR"
```

The semantic functions *deadlocks* and *revivals* are declared as above. Both the functions are defined recursively as below:

```
primrec
" deadlocks(STOP) = (%M. {⟨⟩}d)"
.
.
.
primrec
" revivals(STOP) = (%M. {}r)"
```

If we implemented the code described as above we would get the following error:

```
Cyclic dependency of constants:
*** "CSP_D_semantics.deadlocks" -> "CSP_D_semantics.revivals" ->
"CSP_D_semantics.deadlocks"
*** The error(s) above occurred in definition
"revivals_proc_def":
```

As the definition of *revivals*(P) uses *deadlocks*(P) which in-turn uses the definition of *revivals*(P). This creates a problem called "cyclic dependency of constants" in the implementation. Hence we indirectly define a definition for each semantic function. In Isabelle, mutual recursion like this can be implemented by putting the recursive definitions in one `primrec` block[1].

5.3.2 Two-Phase Construction

In this section, we explain how to encode the semantic functions of the deadlocks and revivals component separately. It is a two-phase construction. In the first phase, we encode definition of deadlocks and revivals simultaneously, as we are getting "cyclic dependency of constants" error if we define the semantic functions separately. In the second phase, we separate the definitions of deadlocks and revivals from the combined function defined in the first phase.

[1]This was suggested by Dr. Christoph Lüth during my viva.

We implement the encoding of the semantic functions of the model $\mathcal{R}$ based on the theory of syntax, the theories on the parallel operator, the hiding operator, the renaming operator, the sequential operator from the reusable part of CSP-Prover. We also use the theory the domain of the failures component `Set_F` from the stable failures model. We import the theories using the keyword `import` in the following code:

```
theory  CSP_R_Semantics
imports  CSP_syntax Domain_R_cpo Set_F
            Trace_par Trace_hide Trace_ren Trace_seq CSP_T_semantics
```

In the first phase, we define the definitions of deadlocks and revivals simultaneously as given below:

```
1. consts
2.   DeadlockRevivals ::
       "('p,'a) proc => ('p => 'a domR) => ('a setD * 'a setR)"
3. primrec
4. " DeadlockRevivals(STOP) = (%M. ( {⟨⟩}d, {}r ))"
5. " DeadlockRevivals(SKIP) = (%M. ( {}d, {}r ) )"
6. " DeadlockRevivals(P [+] Q) = (%M. (
7.   { s .
8.     (( (s :d (fst(DeadlockRevivals(P) M ) ) |
9.      s :d (fst(DeadlockRevivals(Q) M ) )) & s ~=⟨⟩ )
10.      | ( s :d (fst(DeadlockRevivals(P) M ) ) &
11.      s :d (fst(DeadlockRevivals(Q) M ) )) )}d,
12.  {r.   (EX Z a.
13.    r = (⟨⟩, Z, a) &
14.    ((⟨⟩, Z, a) :r snd (DeadlockRevivals P M) |
15.    (⟨⟩, Z, a) :r snd (DeadlockRevivals Q M)) &
16.    (⟨⟩, Z)
17.      :  {(t, X).
18.      (EX a.   (t, X, a) :r snd (DeadlockRevivals P M)) |
19.      t :d fst (DeadlockRevivals P M) & X <=Evset
20.      } &
21.    (⟨⟩, Z)
22.    :  {(t, X).
23.      (EX a.   (t, X, a) :r snd (DeadlockRevivals Q M)) |
24.      t :d fst (DeadlockRevivals Q M) & X <= Evset
25.      }) |
26.    (EX s.   (EX X a.   r = (s, X, a)) &
27.    (r :r snd (DeadlockRevivals P M) |
28 r :r snd (DeadlockRevivals Q M)) & s ~= ⟨⟩) }r))"
```

Line number 2 shows the combined semantic definition of deadlocks and revivals. Now we do not define the mutual-recursive definition functions $\mathit{failures}^b_M(P)$ and $\mathit{failures}_M(P)$ directly like:

```
revivals(P [+] Q) = (%M. ...   (s,Y) :f failures(P) & ...  )
```

Line number 16 to 20 shows the definition of

$$\mathit{failures}^b(P) = \{(s,X) \mid X \subseteq \Sigma \wedge s \in \mathit{Dead}(P)\} \cup \{(s,X) \mid (s,X,a) \in \mathit{Rev}(P)\}$$

in $\mathit{revivals}(P[+]Q)$.

In the second phase, we separate the definition of revivals and deadlocks. We separate them with the following code using the product constructor functions `fst` and `snd`.

```
consts
deadlocks ::  "('p,'a) proc => ('p => 'a domR) => 'a setD"
revivals ::  "('p,'a) proc => ('p => 'a domR) => 'a setR"
defs
deadlocks_def:
    "deadlocks (P) M == ( (fst ( (DeadlockRevivals(P) M ))))"
 revivals_def:
    "revivals (P) M == ( (snd ( (DeadlockRevivals(P) M ))))"
```

We show the mutual semantic clauses for each operators as lemmas in terms of deadlocks and revivals. Below we show a lemma proved for the revivals component of the external choice operator. This also shows the use of *failures*^{b} definition in the lemma revivals_EXTERNAL :

```
"lemma revivals_EXTERNAL : revivals (P [+] Q) =
(%M.{ f.  ( EX Z a.  ( f = (<>,Z,a) &
 ((((⟨⟩,Z,a) :r revivals(P) M | (<>,Z,a) :r revivals(Q) M) &
 ((⟨⟩,Z) : {(t, X). (EX a.  (t, X, a) :r (revivals P M)) |
 t :d deadlocks P M & X <=Evset }
 &(⟨⟩,Z) : {(t, X). (EX a.  (t, X, a) :r (revivals Q M)) |
 t :d deadlocks Q M & X <=Evset } )))) |
 (EX s X a.  f = (s,X, a) & (f :r revivals(P) M
 | f :r revivals(Q) M) & s ~= ⟨⟩)}r)"
```

Henceforth, we can use the above lemma as the semantics for *revivals*$(P \Box Q)$. Similarly for other operators, we prove lemmas like above for both revivals and deadlocks components. From now on, we do not have to use the definition of `DeadlocksRevivals`.

The approached followed to assign the meaning to process names is similar to technique used in other models embedded in CSP-Prover. The definition of semantic function of the stable revivals model is given by `semRf` using the semantic functions of the individual component.

```
consts
 semRf ::  "('p,'a) proc => ('p => 'a domR) => 'a domR"
                          ("[[_]]Rf")
 semRfun ::  "('p => ('p,'a) proc) => ('p => 'a domR)
            => ('p => 'a domR)"          ("[[_]]Rfun")
defs
semRf_def:
 "[[P]]Rf == (%M. (tripleR (traces P (fstR o M)) (deadlocks P M)
(revivals P M)))"
semRfun_def:
 "[[Pf]]Rfun == (%M. %p.  [[Pf p]]Rf M)"
```

The refinement and equivalence relation in the model are defined by the following predicates:

```
consts
 refR ::  "('p,'a) proc => ('p => 'a domR) =>
      ('q => 'a domR) => ('q,'a) proc => bool"
                 ("(0_ /<=R[_,_] /_)" [50,0,0,50] 50)
 eqR ::  "('p,'a) proc => ('p => 'a domR) =>
      ('q => 'a domR) => ('q,'a) proc => bool"
                 ("(0_ /=R[_,_] /_)" [50,0,0,50] 50)
defs
 refR_def :  "P1 <=R[M1,M2] P2 == [[P2]]Rf M2 <= [[P1]]Rf M1"
 eqR_def :  "P1 =R[M1,M2] P2 == [[P1]]Rf M1 = [[P2]]Rf M2"
```

The above definition gives the parameterized refinement and equivalence relation which takes an environment as argument for each process. Now we define the default refinement and equivalence relation.

The meaning for the process names are assigned by fix-point function `semRfix` and it is defined as given below:

```
consts
 semRfix ::  "('p => ('p,'a) proc) => ('p => 'a domR)"
                                        ("[[_]]Rfix")
defs
 semRfix_def:  [[Pf]]Rfix ==
 "(if (FPmode = CPOmode) then (LFP ([[Pf]]Rfun)) else the None)"
```

CSP-Prover has a special keyword called `FPmode` to distinguish the approach. The value `CPOmode` represents CPO based approach. Other values `CMSmode` and `Mixmode` represent the approaches where uniqueness is guaranteed for the solutions to recursive processes. Since we implement the CPO based approach only to assign the meaning to recursive process, we check whether `FPmode` is equal to `CPOMode`. If `FMmode` if equal to `CPOMode`, then we find the least fix point.

CSP-Prover has a special function called the process name function `PNfun`. The constant definition `semRfun` binds the process name P into process `PNfun_Π` P where $P \in \Pi$ and each process $P \in \Pi$ behaves like `PNfun_Π` (P) . For the stable revivals model based on CPO approach, an environment MR_Π is assigned as

```
MR_Π = LFP ([[PNfun_Π]]Rfun
```

```
consts
 MR ::  "('p => 'a domR)"
defs
 MR_def :  "MR == [[PNfun]]Rfix"
```

The semantics of each process is given by `semR` using the environment MR and it is defined as follows:

```
consts
 semR ::  "('p,'a) proc => 'a domR"  ("[[_]]R")
defs
 semR_def :  "[[P]]R == [[P]]Rf MR"
```

`MR` is the fixed point of the function `[[PNfun(p)]]Rfun`. In Chapter 6, after proving continuity of the semantic function we will be able to prove that the semantics of process names is the fixed point process-name function, that is `[[Pf]]Rfun MR = MR`.

Now the default refinement and equivalence relation is defined as below:

```
  "P1 <=R P2" == "P1 <=R[MR,MR] P2"
  "P1 =R P2" == "P1 =R[MR,MR] P2"
```

We prove some lemmas to simplify unfolding the definition of semantic function and prove that two processes are equal if individual components of each process are equal.

```
lemma cspR_eqR_semantics:
 "P : procR & Q : procR ==> ((P =R[M1,M2] Q) =
     ((traces P (fstR o M1) = traces Q (fstR o M2)) &
     (deadlocks P M1 = deadlocks Q M2) &
     (revivals P M1 = revivals Q M2) ))"
```

The above lemma follows from the definition of semantic function and decomposing it. Similarly we prove a lemma `cspR_refR_semantics` for the refinement relation. We group these two lemmas into a single lemma as follows:

```
lemmas cspR_semantics = cspR_eqR_semantics cspR_refR_semantics
```

Similarly we prove another two lemmas where the definition of traces equivalence is applied and it is given below:

```
lemma cspR_cspT_eqR_semantics:
 " P : procR & Q : procR ==> (P =R[M1,M2] Q) =
     ((P =T[fstR o M1,fstR o M2] Q) &
     (deadlocks P M1 = deadlocks Q M2) &
     (revivals P M1 = revivals Q M2))"
```

We prove a similar lemma for refinement relation and group into a single lemma as `cspR_cspT_semantics`.

Chapter 6

Properties proved in CSP-Prover

Contents

In this chapter, we explain some important properties proved in CSP-Prover. Firstly, we prove the type correctness problem or checking the healthiness conditions of the semantic functions with a restriction on the language CSP_{TP} in Section 6.1. Secondly, we discuss continuity of semantic functions proved in CSP-Prover in Section 6.2. Finally, we focus on implementing a proof infrastructure for recursive processes using continuity of semantic functions in Section 6.3.

6.1 Type correctness property for semantic functions

As explained in the introduction chapter, one of the important uses of the encoding a language in a theorem prover is that it allows mechanical verification of the semantic functions. In this section, we explain it in detail. Informally, given the healthy denotations of processes P and Q, we prove that the denotations of $P \mathbin{;} Q$, $P \,\Box\, Q$, $P \,[\![X]\!]\, Q$, $P \setminus X$, etc., are also healthy. Here healthy means that the denotational semantics of P satisfies all the healthiness conditions of the stable revivals model. In CSP-Prover, our aim is to prove that

$$(traces_M(P), deadlocks_M(P), revivals(P)) : domR$$

We prove this by the structural induction on P. The induction step for the external choice operator in Isabelle looks like:

```
(traces_M(P), deadlocks_M(P), revivals_M(P) ): domR ∧
(traces_M(Q), deadlocks_M(Q), revivals_M(Q) ): domR
⇒ (traces_M(P□ Q), deadlocks_M(P □ Q), revivals_M(P □ Q)): domR.
```

6.1.1 Counter example for the type correctness of the renaming operator

It turns out that the type correctness of the renaming operator cannot be proven without any restriction on the language CSP_{TP}. We present a counter example given by M. Roggenbach. If we consider Σ is finite, then this is not a counter example. In CSP-Prover Σ can be arbitrary, hence we have to consider this.

Let $\Sigma = \mathcal{N} \cup \{a, b\}$. Now consider the triple $C = (T_C, D_C, R_C) =$

$$(\{\langle\rangle, \langle a\rangle, \langle b\rangle\},\ \emptyset,\ \{(\langle\rangle, X, a),\ (\langle\rangle, X, b) \mid X \in \mathcal{P}_{fin}(\mathcal{N})\}) \in dom(\mathcal{R})$$

where $\mathcal{P}_{fin}(\mathcal{N})$ is the set of all finite subsets of the natural numbers $\mathcal{N}$.

The triple C is healthy: Trivially, it fulfils **T1**, **D1**, **R1**, and **RRS**. Concerning condition **R2**, let $(\langle\rangle, X, a) \in R_C$, and let $Y \subseteq X$ be a subset of X. As X is finite, so is Y. Therefore by definition $(\langle\rangle, Y, a) \in R_C$. The same argument applies to sets of the form $(\langle\rangle, X, b) \in R_C$. Condition **R3** trivially is true for $a, b \in \Sigma$. Let $n \in \mathcal{N}$. Then $(\langle\rangle, \{n\}, a) \in R_C$ as $\{n\}$ is finite.

Now, consider the relation $Rel \subseteq \Sigma \times \Sigma$ with

$$Rel = \{(a, a)\} \cup \{(n, b) \mid n \in \mathcal{N}\},$$

i.e., a is renamed into a, all natural numbers n are renamed into b, and b is not in the domain of Rel.

Now consider the set $C' = R_C[[Rel]] = \{(\langle\rangle, \emptyset, a)\}$, which is the revivals component of C after applying our renaming Rel. We claim that the healthiness condition **R3** does not hold for C': as $(\langle\rangle, \emptyset, a) \in C'$ and $b \in \Sigma$ we need to obtain $(\langle\rangle, \emptyset, b) \in C'$ or $(\langle\rangle, \{b\}, a) \in C'$, which both is not the case. This shows that we cannot prove the type correctness of renaming operator without any restriction on renaming relation.

If we forbid renaming from infinite elements into a single element or the renaming relation set to be finite, then we can prove the type correctness of the renaming operator. We have proved the type correctness of the renaming operator with an assumption that the renaming relation is finite. In CSP-Prover, it is proven as

```
lemma    RENAMING_setR: "finite r --> {rr.  EX sa t X a aa .  rr =
(t,X,aa) & (sa,[[r]]inv X,a) :r (revivals (P) M) & (sa [[r]]* t)
& (a,aa) :  EventPairSet (r::  ('a  'a) set) } :  setR"
```

6.1.2 A proof for the type correctness of the semantic function

We first give the proof and then explain how we implement it in Isabelle in Section 6.1.3. First we prove two simple lemmas which we will use later in proving the type correctness.

Lemma 6.1: Let $(traces_M(P), deadlocks_M(P), revivals_M(P))$ be a denotational value of P. Let

$$\begin{aligned} failures_M(P) \ = \ & \{(s, X), (s, X \cup \{\checkmark\}) \mid (s, X, a) \in revivals_M(P)\} \\ & \cup\{(s, X) \mid X \subseteq \Sigma^{\checkmark} \wedge s \in deadlocks_M(P)\} \\ & \cup\{(s, X) \mid s \frown \langle\checkmark\rangle \in traces_M(P) \wedge X \subseteq \Sigma\} \\ & \cup\{(s \frown \langle\checkmark\rangle, X) \mid s \frown \langle\checkmark\rangle \in traces_M(P) \wedge X \subseteq \Sigma^{\checkmark}\} \end{aligned}$$

If $(traces_M(P), deadlocks_M(P), revivals_M(P))$ is healthy in the stable revivals model, then we prove $\forall\, s, X, Y.\ (s, X) \in failures_M(P) \land Y \subseteq X \implies (s, Y) \in failures_M(P)$.

Proof. We prove this for the first clauses only as the proofs for the other clauses trivially follow from definition.
Suppose $(s, X) \in failures_M(P)$ such that $(s, X, a) \in revivals_M(P)$ for some a.
Suppose $Y \subseteq X$ be arbitrary. By assumption $(traces_M(P), deadlocks_M(P), revivals_M(P))$ satisfies **R2**. Since $\forall\, s, X, a, Y.\ (s, X, a) \in revivals_M(P) \land Y \subseteq X \implies (s, Y, a) \in revivals_M(P)$, we can conclude that $(s, Y, a) \in revivals_M(P)$. By definition of $failures_M(P)$, we obtain $(s, Y) \in failures_M(P)$. □

The above lemma is embedded inside the following lemma in CSP-Prover, as one of healthiness conditions are closedness property of refusal set.

```
lemma failuresR_setF:
"(t,X). ((EX s.  s ^^ ⟨Tick⟩ :t traces (P) (fstR o M) & t = s ^^
⟨Tick⟩ & noTick s) | (EX a .  (t,X,a) :r revivals (P) M) | ( t :d
deadlocks (P) M) | (EX Y a.  (t,Y,a) :r revivals (P) M & a ~=
Tick & X = insert Tick Y)) : setF"
```

Lemma 6.2: Let $(traces_M(P), deadlocks_M(P), revivals_M(P))$ be a denotational value of P.

Let

$$\begin{aligned} failures^b_M(P) &= \{(s, X) \mid (s, X, a) \in revivals_M(P)\} \\ &\quad \cup\{(s, X) \mid X \subseteq \Sigma^{\checkmark} \land s \in deadlocks_M(P)\} \end{aligned}$$

If $(traces_M(P), deadlocks_M(P), revivals_M(P))$ is healthy in the stable revivals model, then we prove $\forall\, s, X, Y.\ (s, X) \in failures^b_M(P) \land Y \subseteq X \implies (s, Y) \in failures^b_M(P)$.

Proof. The proof is very similar to proof of the previous lemma. □

Theorem 6.3: The semantic functions for all the operators in CSP are healthy provided the renaming relation is finite in the renaming operator.

Proof. We prove the type correctness for the hiding and renaming operators here and proofs for some selected operators are available Appendix A.4. The proofs for other renaming operators are similar.

- $\boxed{\text{case: P } [[R]]}$
  Assuming that $(traces_M(P), deadlocks_M(P), revivals_M(P))$ is healthy,
  we prove that $(traces_M(P\ [[R]]), deadlocks_M(\text{ P } [[R]]), revivals_M(\text{ P } [[R]]\ ))$ are also healthy, where

$$\begin{aligned} traces_M(P[[R]]) &= \{t \mid \exists\, t' \in traces_M(P).\ (t', t) \in [[R]]^*\} \\ deadlocks_M(P[[R]]) &= \{t \mid \exists\, t' \in deadlocks_M(P).\ (t', t) \in [[R]]^*\} \\ revivals_M(P[[R]]) &= \{(s, X, a) \mid \exists\, s, a\,.\, s'R^*s \land a'Ra \land (s', R^{-1}(X), a') \in revivals_M(P)\} \end{aligned}$$

  T1
  We first prove $traces_M(P[[R]])$ is non-empty. $\langle\rangle \in traces_M(P[[R]])$ as $\langle\rangle \in traces_M(P)$ and $(\langle\rangle, \langle\rangle) \in [[R]]^*$. Hence, $traces_M(P[[R]])$ is non-empty.

We prove that $traces_M(P[[R]])$ is prefix closed.
Suppose $t \in traces_M(P[[R]])$ where $t = \langle a_1, a_2, \dots, a_n \rangle$. Then there exist $t' = \langle b_1, b_2, \dots, b_n \rangle$ with $b_i R a_i$ for $1 \le i \le n$ and $t' \in traces(P)$.

Suppose s is a prefix of t. Then $s = \langle a_1, a_2, \dots, a_k \rangle$ such that $k \le n$. Since $traces_M(P)$ is prefix closed, we know that $\langle b_1, b_2, \dots, b_k \rangle \in traces_M(P)$. Thus by definition of $traces(P[[R]])$ with $b_i R a_i$ for $1 \le i \le k$, $s \in traces_M(P[[R]])$. Hence, $traces_M(P[[R]])$ is prefix closed.

D1
Suppose $s \in deadlocks_M(P \,\|[\, R]\|)$. By definition we know that there exists s' with $(s', s) \in [[R]]^*$ such that $s' \in deadlocks_M(P)$. By assumption, we know that $s' \in traces_M(P)$. By definition of $traces_M(P[[R]])$, $s \in traces_M(P[[R]])$.
Since s is an arbitrary, $deadlocks_M(P[[R]]) \subseteq traces_M(P[[R]])$.

R1
Suppose $(s, X, a) \in revivals_M(P \,\|[\, R]\|)$. By definition we know that there exists s' and a' such that $(s', s) \in [[R]]^*$, $(a', a) \in R$, and $(s', R^{-1}(X), a') \in revivals_M(P)$.
By assumption we know that $s' \frown \langle a' \rangle \in traces_M(P)$. Since $(s', s) \in [[R]]^*$, $(a', a) \in R$ and $s' \frown \langle a' \rangle \in traces_M(P)$, by definition of $traces_M(P[[R]])$, by definition of revivals it follows that $s \frown \langle a \rangle \in traces_M(P[[R]])$.
Since $(s, X, a) \in revivals_M(P \,\|[\, R]\|)$ is an arbitrary element, $P \,\|[\, R]\|$ satisfies **R1**.

R2
Suppose $(s, X, a) \in revivals_M(P \,\|[\, R]\|)$. Then there exist s' and a' such that $(s', s) \in [[R]]^*$, $(a', a) \in R$, and $(s', R^{-1}(X), a') \in revivals_M(P)$.
Suppose $Y \subseteq X$ to be an arbitrary set.
We know that $Y \subseteq X$ implies $R^{-1}(Y) \subseteq R^{-1}(X)$. It is clear that
$(s', R^{-1}(X), a') \in revivals_M(P) \wedge R^{-1}(Y) \subseteq R^{-1}(X) \rightarrow (s', R^{-1}(Y), a') \in revivals_M(P)$.
Therefore, $(s', R^{-1}(Y), a') \in revivals_M(P)$ which implies $(s, Y, a) \in revivals_M(P \,\|[\, R]\|)$.
Hence, $revivals_M(P \,\|[\, R]\|)$ satisfies **R2**.

R3
Suppose $(s, X', a) \in revivals_M(P \,\|[\, R]\|)$. Then there exist s', X, a' with $(s', s) \in [[R]]^*$, $(a', a) \in R$ and $X = R^{-1}(X')$ such that $(s', X, a') \in revivals_M(P)$. Suppose b to be an arbitrary event in Σ.

We prove **R3** by applying induction on the size of R.

Induction Base: $\mid R \mid= 0$, i.e., $R = \emptyset$.
It is trivially true as there is no a' such that $(a', a) \in R$. Hence it is a contradiction.

Induction Base: $\mid R \mid= 1$,

Case 1 : $b \notin dom(R^{-1})$. Then $R^{-1}(X' \cup \{b'\}) = R^{-1}(X') = X$, consequently $(s, X' \cup \{b\}, a) \in revivals_M(P \,\|[\, R]\|)$. Hence **R3** holds.

Case 2 : $b \in dom(R^{-1})$. This mean there exists b' such that $b' \in \Sigma$ and $b' \, R \, b$. Since $(s', X, a') \in revivals_M(P)$ is healthy, we know that $(s', X, b') \in revivals_M(P)$ or $(s', X \cup b', a') \in revivals_M(P)$. By definition of $revivals_M(P \,\|[\, R]\|)$, we know that

$(s, X', b) \in revivals_M(P \,\|[R]\|)$ or $(s, X' \cup b, a) \in revivals_M(P \,\|[R]\|)$.
Hence, it satisfies **R3**.

Induction step: $| R |= n + 1$,

Case $1 : b \notin dom(R^{-1})$. Then $R^{-1}(X' \cup \{b'\}) = R^{-1}(X') = X$, consequently $(s, X' \cup \{b'\}, a) \in revivals_M(P \,\|[R]\|)$. Hence, **R3** holds.

Case $2 : b \in dom(R^{-1})$. Then there exists at least one b' with $b' \, R \, b$, i.e. the set

$$B := \{b' \in \Sigma \mid b' \, R \, b\}$$

is non-empty.

Say $B = \{b_1, \ldots, b_{k+1}\}$ and $k + 1 \leq n$. We consider the following two cases.

Case 2.1: $\exists \, b_0 \in B.(s', X, b_0) \in revivals_M(P)$, then we know that $(s, X', b) \in revivals_M(P \,\|[R]\|)$.

Case 2.2: $\forall \, b.b \in B \rightarrow (s', X, b) \notin revivals_M(P)$.

Consider **R3** for b_{k+1} and by induction hypothesis $(s', X \cup (B - \{b_{k+1}\}), a') \in revivals_M(P)$. As **R3** holds for $revivals_M(P)$, we have that
$(s', X' \cup (B - \{b_{k+1}\}), b_{k+1}) \in revivals_M(P) \vee (s', X' \cup B, a') \in revivals_M(P)$.

case 2.2.1: $(s', X' \cup (B - \{b_{k+1}\}), b_{k+1}) \in revivals_M(P)$.
By $(s', X' \cup (B - \{b_{k+1}\}), b_{k+1}) \in revivals_M(P)$, we have $(s', X', b_{k+1}) \in revivals_M(P)$ by **R2**. Hence a contradiction.

case 2.2.2: $(s', X' \cup B, a') \in revivals_M(P)$
By definition of $revivals_M(P \,\|[R]\|)$, $(s, X' \cup \{b\}, a) \in revivals_M(P \,\|[R]\|)$.

Hence R3 holds.
RRS05
Suppose $(s, X', a) \in revivals_M(P \,\|[R]\|)$. As $(s, X', a) \in revivals_M(P \,\|[R]\|)$ satisfies **R2** there exist s', a' with $(s', s) \in [[R]]^*$, and $(a', a) \in R$ such that $(s', R^{-1}(X'), a') \in revivals_M(P)$.

By assumption, we know that $a' \notin R^{-1}(X')$. We have to prove that $a \notin X'$.

By the definition of $R^{-1}(X) = \{a \mid \exists \, b \in X. \, (a, \, b) \in R \vee a = b = \checkmark\}$ we consider two cases

case 1: $a = \checkmark$. By the definition of $revivals_M(P)$, we know that $a \neq \checkmark$. hence it is a contradiction.

case 2: Assume that $a \in X'$. Then by the definition of $R^{-1}(X')$, $a' \in R^{-1}(X')$ as $(a', a) \in R$. This is a contradiction with the assumption. Hence $a \notin X'$.

Therefore **RRS05** holds for $revivals_M(P \,\|[R]\|)$.

- $\boxed{\text{case: } P \setminus X:}$
Assuming that $(traces_M(P), deadlocks_M(P), revivals_M(P))$ is healthy,
we prove that $(traces_M(\text{P} \setminus X), deadlocks_M(\text{P} \setminus X), revivals_M(\text{P} \setminus X))$ is also healthy, where

$$\begin{array}{lcl} traces_M(P \setminus X) & = & \{t \setminus X \mid t \in traces_M(P)\} \\ deadlocks_M(P \setminus X) & = & \{t \setminus X \mid t \in deadlocks_M(P)\} \\ revivals_M(P \setminus X) & = & \{(s \setminus X, Y, a) \mid (s, Y \cup X, a) \in revivals_M(P)\} \end{array}$$

T1
First we prove that $traces_M(P \setminus X)$ is non-empty.
$\langle\rangle \in traces_M(P \setminus X)$ as $\langle\rangle \in traces_M(P)$ and $\langle\rangle \setminus X = \langle\rangle$. Hence $traces_M(P \setminus X)$ is non-empty.

Let $t \in traces_M(P \setminus X)$ where $t = \langle t_1, t_2, \ldots, t_n\rangle$. Then there exists $t' \in traces_M(P)$ with $t' = x_0 \frown \langle t_1\rangle \frown x_1 \frown \langle t_2\rangle \frown x_2 \ldots x_{n-1} \frown \langle t_n\rangle \frown x_n$ where $x_i \in X^*$, for $1 \le i \le n$.

Let $s = \langle t_1, t_2 \ldots t_k\rangle$ be a prefix of t,
then $s' = x_0 \frown \langle t_1\rangle \frown x_1 \frown \langle t_2\rangle \frown x_2 \ldots x_{k-1} \frown \langle t_k\rangle \frown x_k$ is a prefix of t'.

As $traces_M(P)$ is prefix closed, $s' \in traces_M(P)$. By the definition of $traces_M(P \setminus X)$, $(s' \setminus X = s) \in traces_M(P \setminus X)$.
Since s is an arbitrary prefix of t, $traces_M(P \setminus X)$ is prefix closed.

D1
Let $s \in deadlocks_M(P \setminus X)$, then there exists $s' \in deadlocks_M(P)$ with $s = s' \setminus X$.
As $(traces_M(P), deadlocks_M(P), revivals_M(P))$ is healthy, we have $s' \in traces_M(P)$. By definition, this implies that $s \in traces_M(P \setminus X)$.
Therefore, **D1** is fulfilled.

Hence **D1** is satisfied.

R1
Suppose $(s \setminus X, Y, a) \in revivals_M(P \setminus X)$ such that $(s, X \cup Y, a) \in revivals_M(P)$. We know by assumption that $(s, X \cup Y, a) \in revivals_M(P) \longrightarrow s \frown \langle a\rangle \in traces_M(P)$. By definition of $traces_M(P \setminus X)$, we know that $(s \frown \langle a\rangle) \setminus X \in traces_M(P \setminus X)$ which is $((s \setminus X)\frown \langle a\rangle) \in traces_M(P \setminus X)$ as $a \notin X$. Therefore, $(s \setminus X, Y, a) \in revivals_M(P \setminus X)$ implies $((s \setminus X) \frown \langle a\rangle) \in traces_M(P \setminus X)$.
Hence, **R1** is satisfied.

R2
Suppose $(s \setminus X, Y, a) \in revivals_M(P \setminus X)$ and $Z \subseteq Y$. By definition we know that there exist $(s, Y \cup X, a) \in revivals_M(P)$. We also know from hypothesis that
$(s, Y \cup X, a) \in revivals_M(P) \wedge (Z \cup X) \subseteq (Y \cup X) \longrightarrow (s, Z \cup X, a) \in revivals_M(P)$.
From hypothesis $(s \setminus X, Y \cup X, a) \in revivals_M(P \setminus X)$ and $Z \subseteq Y$, it follows that $(s \setminus X, Z, a) \in revivals_M(P \setminus X)$.
Hence, $revivals_M(P \setminus X)$ satisfies **R2**.
R3
Suppose $(s \setminus X, Y, a) \in revivals_M(P \setminus X)$. Hence we know that there exist $(s, Y \cup X, a) \in revivals_M(P)$. We also know from hypothesis that
$(s, Y \cup X, a) \in revivals_M(P) \wedge c \in \Sigma \longrightarrow (s, Y \cup X, c) \in revivals_M(P) \vee (s, X \cup Y \cup \{c\}, a) \in revivals_M(P)$. If $(s, Y \cup X, c) \in revivals_M(P)$, then we know by the definition of $revivals_M(P \setminus X)$ that $(s \setminus X, Y \cup X, c) \in revivals_M(P \setminus X)$.
If $(s, X \cup Y \cup \{c\}, a) \in revival(P)$, then we know by definition of $revivals_M(P \setminus X)$ that

$(s \setminus X, Y \cup \{c\}, a) \in \mathit{revivals}_M(P \setminus X)$.
Hence, $\mathit{revivals}_M(P \setminus X)$ satisfies **R3**.

RRS05
We have to prove that $\forall\ s,\ X\ , a\ .\ (s, Y, a) \in \mathit{revivals}_M(P \setminus X) \rightarrow a \notin Y$. This directly follows from the assumption of **RRS05** on process P and the definition of $\mathit{revivals}_M(P \setminus X)$.

□

6.1.3 Implementation of the type correctness property

The approach followed to prove the type correctness property of the semantic function is similar to the implementation of the domain. We first prove the type correctness for the individual components and then we prove the type correctness for the domain. We have implemented the type correctness for the deadlock component in theory `CSP_R_deadlocks` and the revivals component in theory `CSP_R_revivals`. The type correctness for the domain in theory is implemented in `CSP_R_domain`. The type correctness for the traces component follows from the traces model implemented in CSP-Prover.

We discuss the type correctness of the revivals component only as it includes the two major healthiness conditions **R2** and **R3**. The proofs presented in the previous section are similar to the proof implemented in CSP-Prover. Similarly **T1** has been proved in the domain of the traces component. As we implemented the revivals component as data type `'a setR`, checking type correctness boils down to checking membership like *revivals*$_M$ `(P □ Q) : setR`. We have implemented **R2** and **R3** in the domain of the revivals component, proving *revivals*$_M$ `(P □ Q) : setR` implies that the external operator satisfies the healthiness conditions **R2** and **R3** in the domain of the stable revivals.

For each operators we prove the type correctness of the deadlocks and revivals components. The lemma for the prefix operator is given below:

```
lemma
PREFIX_setD: "f.  (EX s .  f = ⟨ Ev a⟩ ^^ s & s :d D ) : setD"
lemma PREFIX_setR:
"{f.  (EX X. f = (⟨⟩,X,Ev a) & X <= Evset & Ev a ~:X ) | (EX s X
b.  (f = (⟨ Ev a⟩ ^^ s, X,b) & (s,X,b) :r P) ) } :  setR"
```

While proving the process refinements, we frequently need to check `(f :d deadlocks(a -> P) M)`. Hence we unfold the definition of the prefix operator using `PREFIX_setD` as `(EX s . f = ⟨ Ev a⟩ ^^ s & s :d deadlocks(P) M)`. We group all the type correctness lemma for the deadlocks component in a single lemma `in_deadlocks`.

```
lemmas in_deadlocks = in_deadlocks_STOP in_deadlocks_SKIP
                      in_deadlocks_DIV in_deadlocks_PREFIX ...
```

Similarly we do for the revivals components in `in_revivals`.

We are able to prove the type correctness when the renaming relation is finite. But the renaming relation in CSP$_{\mathrm{TP}}$ can be arbitrary. One way is to modify the syntax in the reusable part by creating a process type which contains only processes whose renaming relation is finite. Another way is to create a subset of processes using inductively defined set. We follow the second approach by defining a subset of processes which has only finite renaming relations. In this way, we can reuse all the previous lemmas in theory CSP_Syntax. We start by defining a type of the subset of processes as procR. It is defined as follows:

```
consts
 procR ::  "('p,'a) proc set"
```

We declare the subset by the keyword inductive. It consists of introduction rules.

```
inductive "procR"
intros
 procR_STOP:
 "STOP : procR"

 ...
 procR_Renaming:
 "[| P : procR ; finite r |] ==> (P [[r]]) :  procR"
 ...
```

Isabelle creates a fixed point definition for procR and proves some useful theorem. The introduction rule procR_STOP states that the process STOP is in the set procR and provides the lemma procR.procR_STOP.

$$\mathtt{STOP} \in \mathtt{procR}$$

The rule procR_Renaming states that if P is in procR and the renaming relation r is finite, then (P [[r]]) is in procR and creates the theorem procR.procR_Renaming

$$[[P \in procR;\ finite\ r\ ]] \implies P[[r]] \ \in \mathtt{procR}$$

Similarly for other operators are also defined without any restriction. All the lemmas are grouped in the theorem procR.intros. For convenience, we prove lemmas to show that processes belong to the inductive set with attribute simp in order to apply implication automatically.

```
lemma procR_Renaming[simp]:
 "[| P : procR ; finite r |] ==> (P [[r]]) :  procR"
by (simp add:  procR.intros)
```

The lemma for the type correctness of the external choice operator in the domain of the stable revivals model is

```
lemma EXTERNAL_domR :
"[|(traces(P) (fstR o M), deadlocks(P) M, revivals(P) M ) : domR
; (traces(Q) (fstR o M), deadlocks(Q) M , revivals(Q) M ) : domR
|] ==>
(traces(P [+] Q) (fstR o M), deadlocks(P [+] Q) M,
                          revivals(P [+] Q) M) : domR"
```

The type correctness for the renaming operator is given by the below lemma:

```
lemma RENAMING_domR: "[| finite r ;
(traces(P) (fstR o M), deadlocks(P) M, revivals(P) M ) : domR
|]==>
(traces(P [[r]]) (fstR o M), deadlocks(P [[r]]) M,
                          revivals(P [[r]]) M) : domR"
```

In the above lemma, we also have the assumption "finite r". The type correctness for process P is proved by induction on P in the below lemma:

```
lemma proc_domR_lm[simp]:
"P : procR -->
 (traces(P) (fstR o M), deadlocks(P) M, revivals(P) M) : domR"
```

After applying the command `apply(induct_tac P)`, we will have a subgoal for each operator. For the renaming operator, the subgoal is given as follows :

```
!!proc set.  proc :  procR -->
(traces proc (fstR o M), deadlocks proc M, revivals proc M) :
domR ==>
proc [[set]] :  procR -->
traces (proc [[set]]) (fstR o M), deadlocks (proc [[set]]) M,
revivals (proc [[set]]) M : domR
```

Then, after applying following commands:

```
 apply (intro impI)
 apply (simp)
 apply (erule procR.elims)
 apply (simp_all)
```

we will end-up with the following subgoal:

```
[|(traces P (%u. fstR (M u)), deadlocks P M, revivals P M) :
domR; proc = P & set = r; P : procR; finite r |]
 ==> (traces (P [[r]]) (revivals (P [[r]]) M) : domR
```

Now we can finish off the subgoal with the following command:

```
apply (rule RENAMING_domR, simp)
apply (simp)
```

Similarly, we prove the other subgoals. Theory `CSP_R_domain` also contains lemmas to project out the traces, the deadlock and the revivals component.

```
lemma thdR_semR[simp]:  "P : procR -->
                          thdR [[P]]R = revivals(P) MR"
apply (simp add:  semR_def)
done
```

We also prove a lemma to decompose components of the semantic functions.

```
lemma semRf_decompo:
"P : procR --> (([[P]]Rf M = SF) = ((traces P (fstR o M) = fstR
SF) & (deadlocks P M = sndR SF) & (revivals P M = thdR SF) ))"
```

6.2 Continuity of semantic functions

Another essential property proved in CSP-Prover is continuity of the semantic functions. It is needed in assigning semantics to recursive processes. It is proved in theory `CSP_R_continuous`. It is implemented on the top of theory `CSP_R_domain` which has lemmas for the type correctness, `CSP_T_continuous` which implements continuous property for the traces component, and `Domain_R_cpo` which has proofs to show that the domain of the stable revivals model is a CPO. The reusable part of CSP-Prover on continuity is imported through `CSP_T_continuous`.

```
theory CSP_R_continuous
imports CSP_R_domain Domain_R_cpo CSP_T_continuous
```

In the reusable part of CSP-Prover, it is proved that the definition of continuous given in Definition 2.10 is equivalent to below:

Definition 6.4: Let P and Q be two complete partial orders. Let f be a function from P to Q. f is continuous if and only if whenever $\Delta \subseteq P$ is directed, there exists x such that $x = \bigsqcup \Delta$ and $\bigsqcup\{f(x) \mid x \in \Delta\} = f(\bigsqcup \Delta)$.

This is proved in the following lemma:

```
lemma continuous_iff:
"continuous f = (ALL X. directed X -->
                    (EX x.  ((f x) isLUB (f ` X) & x isLUB X)))"
```

From now on, we use the above characterisation only. First we present the continuity proof for the prefix operators and then we discuss how we implemented this proof. The proofs for other operators are similar.

6.2.1 A Proof of Continuity

In the section, we present a proof of continuity of the prefix operator. The proofs for the other operators are very similar.

Lemma 6.5: If $(traces_M(P), deadlocks_M(P), revivals_M(P))$ is continuous, then $(traces_M(a \rightarrow P), deadlocks_M(a \rightarrow P), revivals_M(a \rightarrow P))$ is continuous.
where
$traces_M(a \rightarrow P) = \{\langle\rangle\} \cup \{\langle a\rangle ^\frown t' \mid t' \in traces_M(P)\}$
$deadlocks_M(a \rightarrow P) = \{\langle a\rangle ^\frown t' \mid t' \in deadlocks_M(P)\}$
$revivals_M(a \rightarrow P) = \{(\langle\rangle, X, a) \mid a \notin X\} \cup \{(\langle a\rangle ^\frown t', X, b) \mid (t', X, b) \in revivals_M(P)\}$

Proof. We prove that the individual components are continuous. We consider only the deadlock component and the revival component as the proof for the traces component has already been proved in CSP-Prover.

Assuming $deadlocks_M(P)$ is continuous, we prove that $deadlocks_M(a \rightarrow P)$ is continuous.

Assuming that for all directed set Δ,
if $deadlocks_{\bigsqcup\Delta}(P) \ = \bigsqcup\{deadlocks_M(P) \mid M \in \Delta\}$ holds,
then we prove that
$deadlocks_{\bigsqcup\Delta}(a \to P) \ = \bigsqcup\{deadlocks_M(a \to P) \mid M \in \Delta\}$ holds.

We know that the least upper bound of $\{deadlocks_M(P) \mid M \in \Delta\}$ is
$\cup\{deadlocks_M(P) \mid M \in \Delta\}$.

Suppose Δ is an arbitrary directed and non-empty set.
Then by the definition of continuity we know that
$deadlocks_{\bigsqcup\Delta}(P) = \ \cup\{deadlocks_M(P) \mid M \in \Delta\}$.

We prove that
$deadlocks_{\bigsqcup\Delta}(a \to P) \ = \ \cup\{deadlocks_M(a \to P) \mid M \in \Delta\}$.

We consider two directions separately.

First, we prove $deadlocks_{\bigsqcup\Delta}(a \to P) \ \subseteq \ \cup\{deadlocks_M(a \to P) \mid M \in \Delta\}$.

Suppose $t \in deadlocks_{\bigsqcup\Delta}(a \to P)$.
By definition of $deadlocks_{\bigsqcup\Delta}(a \to P)$, we know there exist ta such that $t = \langle a\rangle \frown ta$ and $ta \in deadlocks_{\bigsqcup\Delta}(P)$.
By assumption, we know that $deadlocks_{\bigsqcup\Delta}P = \cup\{deadlocks_M(P) \mid M \in \Delta\})$.
By definition of $\cup\{deadlocks_M(P) \mid M \in \Delta\})$, we know there exists δ such that $\delta \in \Delta$ and $ta \in deadlocks_\delta(P)$.

Consider the right hand side.
By definition of $\cup\{deadlocks_M(a \to P) \mid M \in \Delta\}$, we know that
$\exists \ M_1 \ .t \in deadlocks_{M_1}(a \to P) \ \wedge \ M_1 \in \Delta$.
Let $M_1 = \delta$ in $\exists \ M_1 \ .t \in deadlocks_{M_1}(a \to P) \ \wedge \ M_1 \in \Delta$,
then we know that $t \in deadlocks_\delta(a \to P)$ and $\delta \in \Delta$.
By definition of $t \in deadlocks_\delta(a \to P)$, $\exists\, tta.\langle a\rangle \frown tta = t \wedge tta \in deadlocks_\delta(P)$.
Let $tta = ta$, in the above equation then we get $ta \in deadlocks_\delta(P)$.

Therefore, $\forall\, t \ . \ t \in deadlocks_{(\cup\Delta)}(a \to P) \implies t \in \cup\{deadlocks_M(a \to P) \mid M \in \Delta\}$.

Now we consider the reverse direction.
$\cup\{deadlocks_M(a \to P) \mid M \in \Delta\} \ \subseteq \ deadlocks_M(a \to P)\ (\bigsqcup\Delta)$.
Suppose $t \in \cup\{deadlocks_M(a \to P) \mid M \in \Delta\}$.
By definition of $\cup\{deadlocks_M(a \to P) \mid M \in \Delta\}$, we know that there exists δ such that $t \ \in deadlocks_\delta(a \to P) \ \wedge \ \delta \in \Delta$.
By definition of $deadlocks_\delta(a \to P)$, there exist ta such that
$\langle a\rangle \frown ta \in deadlocks_\delta(a \to P)$, $t = \langle a\rangle \frown ta$, and $ta \in deadlocks_\delta(P)$.

Now consider the right hand side
By definition of $deadlocks_M(a \to P)$, $\exists \ tta.t = \langle a\rangle \frown tta \wedge tta \in deadlocks_{\bigsqcup\Delta}(P)$.
Let $tta \ = \ ta$, in $\exists \ tta.tt \ = \ \langle a\rangle \frown tta \ \wedge \ tta \ \in \ deadlocks_{\bigsqcup\Delta}(P)$. Then we know that $t = \langle a\rangle \frown ta \in deadlocks_{\bigsqcup\Delta}(P)$.
By assumption $deadlocks_{\bigsqcup\Delta}P \ = \ \cup\{deadlocks_M(P) \ \mid \ M \ \in \ \Delta\})$, we know that $\exists\,\delta_0 \ .\delta_0 \in \Delta \wedge ta \in deadlocks_{\delta_0}(P)$.
Let $\delta_0 = \delta$ in $\exists\,\delta_0 \ .\delta_0 \in \Delta \wedge ta \in deadlocks_{\delta_0}(P)$, then we obtain

$\delta \in \Delta \wedge ta \in deadlocks_\delta(P)$.
Therefore $\forall\, t\,.\; t \in \cup\{deadlocks_M(a \rightarrow P) \mid M \in \Delta\} \Longrightarrow t \in deadlocks_M(a \rightarrow P)\ (\bigsqcup \Delta)$.

Now we prove continuity for the revivals component.
$revival_M(a \rightarrow P) = \{(\langle\rangle, X, a) \mid a \notin X\} \cup \{(\langle a \rangle ^\frown t', X, b) \mid (t', X, b) \in revivals_M(P)\}$.

Assuming that for all directed set Δ,
if $revivals_{\bigsqcup\Delta}(P) = \bigsqcup\{revivals_M(P) \mid M \in \Delta\})$ holds,
then we prove that
$revivals_{\bigsqcup\Delta}(a \rightarrow P) = \bigsqcup\{revivals_M(a \rightarrow P) \mid M \in \Delta\})$ holds.

We know that the least upper bound of $\{revivals_M(P) \mid M \in \Delta\}$ is $\cup\{revivals_M(P)M \mid M \in \Delta\}$.

We prove that $revivals_{\bigsqcup\Delta}(a \rightarrow P) = \cup\{revivals_M(a \rightarrow P) \mid M \in \Delta\}$.

Consider the first clause $\{(\langle\rangle, X, a) \mid a \notin X\}$. The first clauses is trivial as $\{(\langle\rangle, X, a) \mid a \notin X\} \subseteq revivals_{\bigsqcup\Delta}(a \rightarrow P)$ and $\{(\langle\rangle, X, a) \mid a \notin X\} \subseteq \cup\{revivals_M(a \rightarrow P) \mid M \in \Delta\}$ as Δ is non-empty. Thus, $revivals_{\bigsqcup\Delta}(a \rightarrow P) = \cup\{revivals_M(a \rightarrow P) \mid M \in \Delta\}$.

Consider the second clause $\{(\langle a \rangle ^\frown t', X, b) \mid (t', X, b) \in revivals_M(P)\}$.

First we prove that $revivals_{\bigsqcup\Delta}(a \rightarrow P) \subseteq \cup\{revivals_M(a \rightarrow P) \mid M \in \Delta\}$.
Suppose $r \in revivals_{\bigsqcup\Delta}(a \rightarrow P)$.
By definition of $revivals_{\bigsqcup\Delta}(a \rightarrow P)$, we know there exist t, X, and b such that $r = (\langle a \rangle ^\frown t, X, b)$, $(\langle a \rangle ^\frown t, X, b) \in revivals_{\bigsqcup\Delta}(a \rightarrow P)$ and $(t, X, b) \in revivals_{\bigsqcup\Delta}(P)$.
By assumption, we know that $revivals_{\bigsqcup\Delta}P = \cup\{revivals_M(P) \mid M \in \Delta\})$.
By definition of $\cup\{revivals_M(P) \mid M \in \Delta\})$, we know there exists δ such that $\delta \in \Delta$ and $(t, X, b) \in revivals_\delta(P)$.

Consider the right hand side, by definition of $\cup\{revivals_M(a \rightarrow P) \mid M \in \Delta\}$, we know that $\exists\ \delta_1\,.(\langle a \rangle ^\frown t, X, b) \in revivals_{\delta_1}(a \rightarrow P) \wedge \delta_1 \in \Delta$.
Let $\delta_1 = \delta$ in $\exists\ \delta_1\,.(\langle a \rangle ^\frown t, X, b) \in revivals_{\delta_1}(a \rightarrow P) \wedge \delta_1 \in \Delta$.
By definition of $(\langle a \rangle ^\frown t, X, b) \in revivals_\delta(a \rightarrow P)$, we obtain $(t, X, b) \in revivals_\delta(P)$.
Therefore $\forall\, r\,.\; r \in revivals_{\bigsqcup\Delta}(a \rightarrow P) \Longrightarrow r \in \cup\{revivals_M(a \rightarrow P) \mid M \in \Delta\}$.

Now we consider the reverse direction.
$\cup\{revivals_M(a \rightarrow P) \mid M \in \Delta\} \subseteq revivals_{\bigsqcup\Delta}(a \rightarrow P)$.
Suppose $(t, X, b) \in \cup\{revivals_M(a \rightarrow P) \mid M \in \Delta\}$.
By definition of $\cup\{revivals_M(a \rightarrow P) \mid M \in \Delta\}$, we know that there exists δ such that $(t, X, b) \in revivals_\delta(a \rightarrow P) \wedge \delta \in \Delta$.
By definition of $revivals_\delta(a \rightarrow P)$, there exist ta, X and b such that $(\langle a \rangle ^\frown ta, X, b) \in revivals_\delta(a \rightarrow P)$, $t = \langle a \rangle ^\frown ta$, and $(ta, X, b) \in revivals_\delta(P)$.

Now consider the right hand side
By definition of $revivals_{\bigsqcup\Delta}(a \rightarrow P)$, we know that
$\exists\ tta\ X_0\ b_0.(t, X_0, b_0) \in revivals_{\bigsqcup\Delta}(P) \wedge t = \langle a \rangle ^\frown tta$.
Let $X_0 = X$, $tta = ta$ and $b_0 = b$ in the above equation,
then we get that $(\langle a \rangle ^\frown ta, X, b) \in revivals_{\bigsqcup\Delta}(P)$.
By assumption $revivals_{\bigsqcup\Delta}P = \cup\{revivals_M(P) \mid M \in \Delta\})$.
By definition of $\cup\{revivals_M(P) \mid M \in \Delta\})$,

we know that $\exists\,\delta_0\ .\delta_0 \in \Delta \wedge (ta, X, b) \in \mathit{revivals}_{\delta_0}(P)$.
Let $\delta_0 = \delta$ in $\exists\,\delta_0\ .\delta_0 \in \Delta \wedge (ta, X, b) \in \mathit{revivals}_{\delta_0}(P)$, then we get $\delta \in \Delta \wedge (ta, X, a) \in \mathit{revivals}_{\delta}(P)$.
Therefore $\forall\ r\ .\ r \in \cup\{\mathit{revivals}_M(a \rightarrow P) \mid M \in \Delta\} \implies r \in \mathit{revivals}_{\bigsqcup\Delta}(a \rightarrow P)$.

□

6.2.2 Implementing Continuity of the Semantic function in CSP-Prover

In this section, we explain briefly about implementing continuity of the semantic function of the stable revivals model in CSP-Prover. First we prove that a triple is continuous if and only if each of the individual components are continuous.

```
lemma continuous_domR_decompo:
"ALL x.  (f x, g x, h x) :  domR ==>
continuous (%x.  (tripleR (f x) (g x) (h x))) =
(continuous f & continuous g & continuous h)"
```

We prove the above lemma using the following two lemmas which proves that each direction implies another. We need only one direction to prove the continuity of the semantic function. We generalise the lemma by proving equivalence.

```
lemma continuous_domR:
 "[| ALL x.  (f x, g x, h x):  domR ; continuous f ; continuous g
; continuous h |]
 ==> continuous (%x.  (tripleR ( f x) (g x) (h x)) )"

lemma continuous_domR_decompo_only_if:
"[| ALL x.  (f x, g x, h x) :  domR; continuous (%x.  ( tripleR
(f x) (g x) (h x))) |]
==> continuous f & continuous g & continuous h"
```

The above two lemmas follows by the lemma `pair_continuous` which is proved in reusable theory `CPO_pair` and continuous property of the domain functions `Abs_domR`.

We also have to prove that the induced component *failuresR* is continuous when proving continuity of the semantic functions of the individual components. We prove *deadlocks*, *revivals* and *failuresR* components are continuous simultaneously as we need to assume *deadlocks* and *revivals* are continuous to prove the parallel operators are continuous.

This is proved in the below lemma:

```
lemma continuous_failuresR:
 "P : procR --> continuous (failuresR P) & continuous (revivals
P) & continuous (deadlocks P)"
```

We prove continuity of the induced component *failuresR* for each operator separately like,

```
lemma continuous_failuresR_Parallel:
 "[| continuous (revivals P) ; continuous (revivals Q);
continuous (deadlocks P); continuous (deadlocks Q) |] ==>
 continuous (failuresR (P |[X]| Q))"
```

We prove the semantic function is continuous by the following lemma:

```
lemma continuous_semRf: "P : procR ==> continuous [[Pf]]Rf"
apply (simp add:  semRf_def)
apply (simp add:  continuous_domR_decompo)
```

Applying the above two commands will produce the goal as

```
1.  P : procR ==>
 continuous (%M::'a => 'b domR. traces P (fstR o M)) & continuous
(deadlocks P) & continuous (revivals P)
```

We prove that the traces component is continuous by using continuity property of the traces model. Now we finish the proof using the lemma `continuous_failuresR` by the following commands:

```
apply (simp add:  continuous_traces_fstR)
by (simp add:  continuous_failuresR)
```

Then continuity property of process function is proven by the following lemma:

```
lemma          continuous_semRfun:  "(ALL i .  Pf i :  procR) ==>
continuous [[Pf]]Rfun"
```

In the next section, we will look how continuity property will be used in providing proof a infrastructure for the recursive processes.

6.3 Proof Infrastructure for Recursive processes

In this section, we explain the implementation of a proof infrastructure for recursive processes. Recursive processes are central to reason about any non-trivial process. We have seen in Chapter 2 that if a function f is continuous, then the function f has solutions using Tarski Fixed Point theorem. The implementation of Tarski Fixed Point theorem is available as a reusable part of CSP-Prover in theory `CPO`. We use the continuity property proved in the previous section and theory `CPO` to assign the semantics to recursive processes.

The important theorems required for recursive processes are unwind laws and lemmas to simplify the substitution of the process name by another process in CSP. These are two important lemmas which are more often used in proving the refinement and equivalence relation involving recursive processes. The unwind law is

$$\forall\, p \in \Pi\,.\,[\$p]R \;=\; [PNfun_{\Pi}(p)]R$$

In CSP-Prover, the type of $\$p$ is `('p, 'a) proc`, the type of p is `'p` and Π is a set of process names.

The substitution of process name p which is of type $'p_1$ by a process `f(p)` is done by the operator <<. In CSP-Prover, f is a function of type `f :: `$'p_1$` ==> (`$'p_2$`, 'a) proc`. It is implemented in theory `CSP_Syntax`. Another important lemma that needs to be proven is

$$\forall\, f\,.(\forall\, p.\; [(PNfun\, p) \;<<\; f\,]R = [f\, p]R \;\Longrightarrow\; [\$p]R \;\subseteq\; [f\, p]R$$

In the rest of this section, we see how the above two important lemmas have been proven for the stable revivals model in CSP-Prover and a small example to understand its application. The implementation is similar to the other models embedded in CSP-Prover, but the proofs are slightly different as for every process P membership in set `procR` is assumed. First, we look at the unwind law.

The existence of the least fixed point for process-name function follows directly using Tarski fixed point theorem from theory `CPO` and continuity property of the semantic functions. It is proven in the lemma

```
lemma semR_hasLFP_cpo:
"(ALL i .  Pf i :  procR) & Pf = PNfun ==> [[Pf]]Rfun hasLFP"
```

That the semantic meaning for process names are equal to the least fixed point of process-name function are proven by the following two lemmas:

```
lemma semR_LFP_cpo:
 "[| Pf = PNfun ; (ALL i .  Pf i :  procR); FPmode = CPOmode |]
==> [[$p]]R = LFP [[Pf]]Rfun p"

lemma semR_LFP_fun_cpo:
 "[| Pf = PNfun ; (ALL i .  Pf i :  procR); FPmode = CPOmode |]
 ==> (%p.  [[$p]]R) = LFP [[Pf]]Rfun"
```

We need to assume that for all process names `p`, `Pf  p` is in the set `procR`. The proof follows from the definition of fix-point function `semRfix` and the semantic definition of process names. The unwind law for CPO based approach is proven by the following lemma:

```
lemma ALL_cspR_unwind_cpo:
 "[| Pf = PNfun ; (ALL i .  Pf i :  procR); FPmode = CPOmode |]
==> ALL p.  ($p =R Pf p)"

lemma cspR_unwind_cpo:
 "[| Pf = PNfun ; (ALL i .  Pf i :  procR); FPmode = CPOmode |]
==> $p =R Pf p"
```

We prove the above lemmas using fix-point function `semRfix`, and using existence and definition of the least fixed point from theory `CPO`.

We often need to replace a process name by another process while proving the refinement and equivalence relation involving recursive processes. First, we prove that

```
lemma semR_subst:
"(ALL i .  f i :  procR ) & P : procR & P<<f :  procR ==>
[[P<<f]]R = [[P]]Rf (%q.  [[f q]]R) "

lemma semR_subst_semRfun:
"(ALL i .  f i :  procR ) & (ALL x.  ((Pf x) :  procR)) & (ALL x.
((Pf x) << f) :  procR) ==>
 (%q.  [[ (Pf q)<<f ]]R) = ([[Pf]]Rfun (%q.  [[f q]]R))"
```

We prove this by induction on the structure of the syntax. The proof follows directly from the definition of << and semantic definition of the individual components. For all the induction step which does not have a process name, the proof follows trivially from the definition of <<. For the step which has a process name, the proof follows from the semantic definition of individual components. To understand a process which has a process name, consider the example

```
[ $P << g ]R = [ $P ]Rf (g P).
```

After the substitution of the process $\$P$ by definition of <<, we get $[g\,P]R = [\$P]Rf\,(g\,P)$. By definition of `[ $P ]Rf ( g P)`, `[ g P ]R`, and semantic definition of the individual components, both sides are equal. Hence the proof for the first lemma follows. Then we prove the second lemma which follows from the first lemma.

We prove an introduction rule for fixed point induction to replace the process name on the right hand side.

```
lemma cspR_fp_induct_cpo_ref_right:
"[| Pf = PNfun ; FPmode = CPOmode ; Q <=R f p;
 !!  p.  f p <=R (Pf p)<<f; (ALL i .  Pf i :  procR); (ALL i .
f i :  procR); (ALL x.  ((Pf x) << f) :  procR) |] ==> Q <=R $p"
```

`cspR_fp_induct_cpo_ref_right` is useful to prove the refinement relation which has process name on the right hand side. Similarly `cspR_fp_induct_cpo_ref_left` is useful for process name on the left hand side.

To prove equivalence relation which has process name, the following substitution lemma is proven.

```
lemma cspR_greatest_cpo:
"[| Pf = PNfun ; (ALL i .  Pf i :  procR); FPmode = CPOmode ;
(ALL i .  f i :  procR);
(ALL x.  ((Pf x) << f) :  procR); ALL p.  (Pf p) << f =R f p |]
==> f p <=R $p"
```

The proof follows from the definition of the least fixed point of function and the substitution of process names by processes. If we had implemented the Complete Metric Space (CMS) based approach, then we would have got `[f p]R = [$p]R` instead of `[$P]R` $\subseteq$ `[f p]R`.

Consider an example of process which has process name to illustrate the unwind law `cspR_unwind_cpo`. Let $\Sigma = \{a, b\}$ and $\Pi = \{P, Q\}$. Let

$$P = a \rightarrow b \rightarrow \mathtt{STOP} \sqcap a \rightarrow c \rightarrow \mathtt{STOP}$$

and

$$Q = a \rightarrow (b \rightarrow \mathtt{STOP} \sqcap c \rightarrow \mathtt{STOP}).$$

We prove that `P =R Q`. In CSP-Prover, Π and Σ should be declared before it is used. It is declared by the following commands:

```
datatype Event = a | b | c
datatype Name = P | Q
```

Process names P and Q are defined by the function `procPfun` using the keyword `primrec` below:

```
consts
 procPfun ::  "PName=> (PName, Event) proc"
primrec
 "procPfun P = a -> b -> STOP |~| a -> c -> STOP"
 "procQfun Q = a -> (b -> STOP |~| c -> STOP)"
```

The function `procPfun` is defined to be the process name function `PNfun` using `overload` keyword by the following command. The keyword `overload` tells Isabelle that `procPfun` is an instance of the declared type `'p => ('p, 'a) proc` of `PNfun`.

```
defs (overloaded)
 set_procPfun_def [simp]:  "PNfun == procPfun"
```

To compare two processes P and Q, we map process name P to process $\$Q$ and it is defined by function `map_Pproc_to_Pproc` using the keyword `primrec`.

```
consts
 map_Pproc_to_Pproc ::  "PName => (PName, Event) proc"
primrec
 "map_Pproc_to_Pproc P = $Q"
defs
 FPMode_def [simp]:  "FPmode == CPOmode"
```

We now give the constant definitions for processes $\$P$ and $\$Q$ as

```
consts
 procP ::  "(PName, Event) proc"
defs
 procP_def :  "procP == $P"
consts
 procQ ::  "(PName, Event) proc"
defs
 procQ_def :  "procQ == $Q"
```

Now we prove that `procQ =R procP` by the following lemma:

```
[| ALL i::PName.  procPfun i :  procR; ALL i::PName.
map_Pproc_to_Pproc i :  procR |] ==> procQ =R procP
```

Applying the definition of `procQ` and `procP`, we get the following subgoal:

```
[| ALL i::PName.  procPfun i :  procR; ALL i::PName.
map_Pproc_to_Pproc i :  procR |] ==> $Q =R $P
```

To apply the unwind law on the left hand side of the equation, we separate it by the command `apply(rule cspR_rw_left)`. We get the following two subgoals:

```
  1.  [| ALL i::PName.  procPfun i :  procR; ALL i::PName.
map_Pproc_to_Pproc i :  procR |] ==> $Q =R ?P2.0
  2.  [| ALL i::PName.  procPfun i :  procR; ALL i::PName.
map_Pproc_to_Pproc i :  procR |] ==> ?P2.0 =R $P
```

Now we apply the unwind law using the commands `apply (rule cspR_unwind_cpo)` and `apply simp_all)`. We get the following subgoal:

```
[|ALL i::PName.  procPfun i :  procR;
ALL i::PName.  map_Pproc_to_Pproc i :  procR |] ==>
 a -> b -> STOP |~| a -> c -> STOP =R $P
```

Similarly after the unwind law on the right hand side for $\$P$, we get

```
[| ALL i::PName.  procPfun i :  procR; ALL i::PName.
map_Pproc_to_Pproc i :  procR |] ==>
a -> b -> STOP |~| a -> c -> STOP =R
a -> (b -> STOP |~| c -> STOP)"
```

Now the subgoal is proved using the semantic definition of the stable revivals model. In the next chapter, we will discuss another example involving recursive processes which uses process names.

Chapter 7

The Model $\mathcal{R}$ at work

Contents

In Chapter 6, we looked at the implementation of the semantic functions of the stable revivals model and properties proved in CSP-Prover. In this chapter, we demonstrate how to prove algebraic laws and refinements involving recursive processes using the properties proved in Chapter 6. In Section 7.1, we prove and disprove some algebraic laws. In Section 7.2, we give an example for recursive processes to demonstrate the application of our implementation.

The step laws for stop, the external choice operator, and the renaming operator fail. Counter examples for these step laws are given in Section 7.1.2. By modifying the semantics for the prefix choice operator and the renaming operator, we prove the step laws for the failed operators.

7.1 Algebraic Laws

All the CSP models support reasoning about processes through denotational semantics. Algebraic laws can be proved from the semantic functions of these models. An algebraic law is the statement that two expressions, involving some operators and identifiers representing arbitrary processes are equal [Ros98]. We can prove equivalence between CSP processes induced by various denotational models of CSP. For some models of CSP, it is also possible to prove the equality and refinement between processes without using denotational semantic function this means that we can prove whether any two processes are equivalent or not using algebraic laws only. Algebraic laws capture the algebraic properties of operators. They can be used to measure the relative strengths of models in CSP and acts as a sanity test for any proposed semantic model.

CSP laws also play an important role in verification of systems. It is also useful in simplifying complex processes before being used in model checking tools. Tools, like FDR, exploit the CSP laws to reduce the specification to a normal form before embarking on a proof of correctness of an implementation [Hoa06]. We can classify the equality laws into two types

- basic laws, and
- step laws.

In CSP-Prover, algebraic laws are derived from the semantic functions and hence proving these laws in a sound way. In the first section, we explain the basic laws which we have proved and then we explain the important step laws proved in the stable revivals model.

7.1.1 Basic laws

In this section, we briefly explain the steps involved in proving basic laws in the stable revivals model. Basic laws include idempotent, associativity, distributivity, symmetry and commutativity of CSP operators. These laws are shown to be equal by the denotational semantic function and are easy to prove in CSP-Prover. We verified in CSP-Prover all the laws shown Figure 7.1. In the figure, we left out the condition `P : procR` intentionally. For example, the idempotence of the external choice operator

$$P \Box P =_{\mathcal{R}} P \qquad (\Box\text{-idem})$$

is proved as

$$P : procR \Longrightarrow P \Box P =_{\mathcal{R}} P$$

by the proof shown in Figure 7.2.

The basic law $\Box$-*idem* holds mostly in all the CSP models except the stable ready model. As an example, we prove that $P \Box P =_{\mathcal{R}} P$. The proof is similar to the semantic proof as described in the CSP-Prover user manual. We need to prove that the denotation of process on the left hand side is equal to the denotation of process on the right hand side. In CSP-Prover for the stable revivals model, the lemma for the idempotent law of the external operator is given below:

```
lemma cspR_External_choice_idem_p:
        "P : procR ==>(P [+] P) =R[M,M] P"
```

First we prove the above lemma, and then we prove `P : procR ==> P [+] P =R P`. Figure 7.2 shows the screen-shot of the lemma. We need to assume that P is in the set of processes *procR*. It takes the same environment M as argument on both sides. After applying definition of `=R` and decomposing the triple using the command `apply(simp add: cspR_cspT_semantics)`, we get the following:

```
 1. P : procR ==> deadlocks (P [+] P) M = deadlocks P M &
revivals (P [+] P) M = revivals P M
```

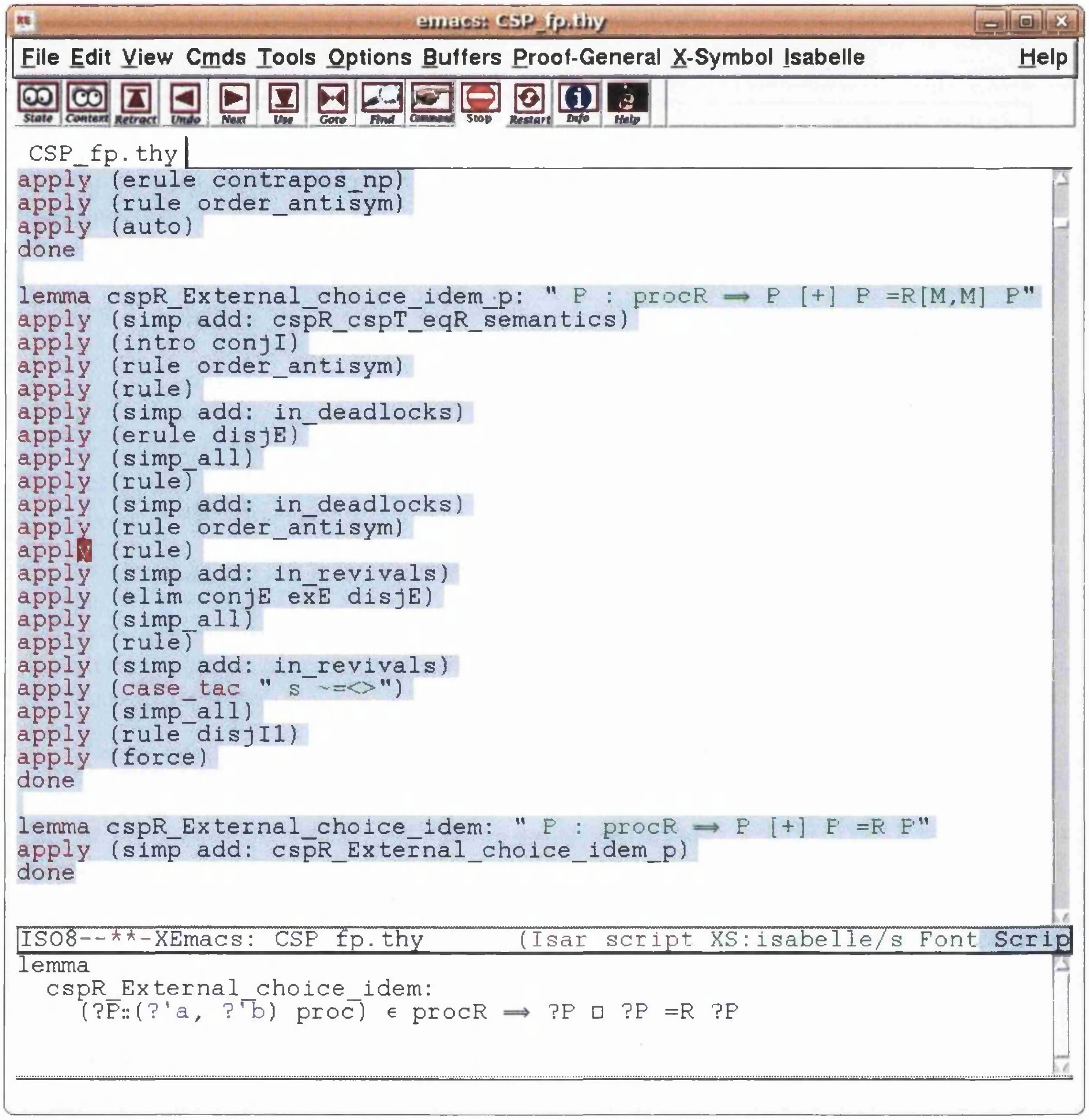
In the traces model, `P [+] P =T[fstR o M,fstR o M] P` is already proven and hence, it is proven automatically and not displayed. We prove that individual components are equal. First we prove that `(deadlocks (P [+] P) M = deadlocks P M )`. By applying anti-symmetric rule using the commands `apply (rule order_antisym)`, we get the two subgoals for `(deadlocks (P [+] P) M = deadlocks P M )` as

```
 1. P : procR ==> deadlocks (P [+] P) M <= deadlocks P M
 2. P : procR ==> deadlocks P M <= deadlocks (P [+] P) M
 3. P : procR ==> revivals (P [+] P) M = revivals P M
```

"cspR_basic"

$P \mathrel{\Box} P =_{\mathcal{R}} P$	($\Box$-idem)
$P \sqcap P =_{\mathcal{R}} P$	($\sqcap$-idem)
$P \mathrel{\Box} Q =_{\mathcal{R}} Q \mathrel{\Box} P$	$\Box$-sym
$P \sqcap Q =_{\mathcal{R}} Q \sqcap P$	$\sqcap$-sym
$P \parallel[X]\parallel Q =_{\mathcal{R}} Q \parallel[X]\parallel P$	($\parallel[X]\parallel$-sym)
$P \mathrel{\Box} (Q \mathrel{\Box} R) =_{\mathcal{R}} (P \mathrel{\Box} Q) \mathrel{\Box} R$	($\Box$-assoc)
$P \sqcap (Q \sqcap R) =_{\mathcal{R}} (P \sqcap Q) \sqcap R$	($\sqcap$-assoc)
$(P \sqcap Q) \mathrel{\Box} R =_{\mathcal{R}} (P \mathrel{\Box} R) \sqcap (P \mathrel{\Box} R)$	($\Box$-$\sqcap$-dist)
$\mathtt{STOP} \parallel[X]\parallel \mathtt{STOP} =_{\mathcal{R}} \mathtt{STOP}$	(STOP-$\parallel[X]\parallel$)
$?\,x : A \rightarrow (P(x) \sqcap Q(x)) =_{\mathcal{R}} (?\,x : A \rightarrow P(x)) \sqcap (?\,x : A \rightarrow Q(x))$	(prefix-choice-dist)
$\begin{array}{l} ((?\,x : A \rightarrow P(x)) \sqcap (?\,x : B -> Q(x))) =_{\mathcal{R}} \\ \quad ?\,x : A \rightarrow (\mathtt{IF}\,(x \in A \cap B)\ \mathtt{THEN}\ P(x) \sqcap Q(x) \\ \qquad \mathtt{ELSE}\ (\mathtt{IF}\ x \in A\ \mathtt{THEN} P(x)\ \mathtt{ELSE}\ Q(x))) \\ \qquad\quad \sqcap \\ \quad ?\,x : B \rightarrow (\mathtt{IF}\,(x \in A \cap B)\ \mathtt{THEN}\ P(x) \sqcap Q(x) \\ \qquad \mathtt{ELSE}\ (\mathtt{IF}\ x \in A\ \mathtt{THEN} P(x)\ \mathtt{ELSE}\ Q(x))) \end{array}$	(generalised-$\sqcap$-dist)

Figure 7.1: CSP Basic laws


emacs: CSP_fp.thy
File Edit View Cmds Tools Options Buffers Proof-General X-Symbol Isabelle Help
State Context Retract Undo Next Use Goto Find Command Stop Restart Info Help
CSP_fp.thy

```
apply (erule contrapos_np)
apply (rule order_antisym)
apply (auto)
done

lemma cspR_External_choice_idem_p: " P : procR ⟹ P [+] P =R[M,M] P"
apply (simp add: cspR_cspT_eqR_semantics)
apply (intro conjI)
apply (rule order_antisym)
apply (rule)
apply (simp add: in_deadlocks)
apply (erule disjE)
apply (simp_all)
apply (rule)
apply (simp add: in_deadlocks)
apply (rule order_antisym)
apply (rule)
apply (simp add: in_revivals)
apply (elim conjE exE disjE)
apply (simp_all)
apply (rule)
apply (simp add: in_revivals)
apply (case_tac " s ~=<>")
apply (simp_all)
apply (rule disjI1)
apply (force)
done

lemma cspR_External_choice_idem: " P : procR ⟹ P [+] P =R P"
apply (simp add: cspR_External_choice_idem_p)
done
```

```
ISO8--**-XEmacs: CSP_fp.thy        (Isar script XS:isabelle/s Font Scrip
lemma
  cspR_External_choice_idem:
    (?P::(?'a, ?'b) proc) ∈ procR ⟹ ?P □ ?P =R ?P
```

Figure 7.2: Screen-shot: Proof for Idempotence law of the external choice operator

By applying the command `apply (rule)` to the first goal, we get

```
1.  !!s::'b trace.  [| P : procR; s :d deadlocks (P [+] P) M |]
==> s :d deadlocks P M
```

Now we apply the command `apply (simp add: in_deadlocks)` to unfold the definition of `deadlocks (P [+] P) M` and prove the subgoal using set theory. `in_deadlocks` is proved in theory `CSP_R_deadlocks`. We prove the reverse direction:

```
 !!  s::'b trace.  [[ P : procR; s :d deadlocks P M ]] ==> s :d
deadlocks (P [+] P) M
```

Similarly, we prove equivalence between the revivals component of processes `P [+] P` and `P` using `apply (simp add: in_revivals)`. `in_revivals` is proved in theory `CSP_R_revivals`. We then prove the default equivalence relation by the following lemma:

```
lemma cspR_External_choice_idem:  " P : procR ==> P [+] P =R P"
apply (simp add: cspR_External_choice_idem_p)
```

Given processes P, Q and R, we prove that the external choice operator distributes over the internal choice operator by the following lemma:

```
lemma ExternalDistributesOverInternal:
" P : procR & Q : procR & R : procR ==> (P |~| Q) [+] R =R[M,M]
(P [+] R) |~| (Q [+] R)."
```

The proof is not as easy as idempotent of internal choice operator.

It is observed in [Ros07] that almost all of the standard algebraic laws of CSP hold and the internal choice operator does not distribute over the external choice operator (□-⊓-dist).

$$(P \sqcap Q) \sqcap R \neq_{\mathcal{R}} (P \sqcap R) \square (P \sqcap R) \qquad (\sqcap - \square - dist)$$

By the definition of semantic function for the revivals component, we get $revivals_M((a \rightarrow \mathrm{STOP} \square b \rightarrow \mathrm{STOP}) \sqcap \mathrm{STOP}) = \{(\langle\rangle, \{\}, a), (\langle\rangle, \{\}, b)\}$.

On the right hand side, we find $failures^b$ to understand it
$failures^b((a \rightarrow \mathrm{STOP}) \sqcap \mathrm{STOP}) = \{(\langle\rangle, \{a, b\}), (\langle\rangle, \{a\}), (\langle\rangle, \{b\}), (\langle\rangle, \{\}),$
$(\langle a\rangle, \{a, b\}), (\langle a\rangle, \{a\}), (\langle a\rangle, \{b\}), (\langle a\rangle, \{\})\}$ and
$failures^b((b \rightarrow \mathrm{STOP}) \sqcap \mathrm{STOP}) = \{(\langle\rangle, \{a, b\}), (\langle\rangle, \{a\}), (\langle\rangle, \{b\}), (\langle\rangle, \{\}),$
$(\langle b\rangle, \{a, b\}), (\langle b\rangle, \{a\}), (\langle b\rangle, \{b\}), (\langle b\rangle, \{\})\}$.

By the definition of semantic functions for the revivals component, we obtain
$revivals_M((a \rightarrow \mathrm{STOP}) \sqcap \mathrm{STOP}) = \{(\langle\rangle, \{b\}, a), (\langle\rangle, \{\}, a)\}$ and
$revivals_M((b \rightarrow \mathrm{STOP}) \sqcap \mathrm{STOP}) = \{(\langle\rangle, \{a\}, b), (\langle\rangle, \{\}, b)\}$.
The revivals component for the right hand side is $\{(\langle\rangle, \{b\}, a), (\langle\rangle, \{\}, a), (\langle\rangle, \{a\}, b), (\langle\rangle, \{\}, b)\}$.
Thus $(P \square Q) \sqcap R \neq_{\mathcal{R}} (P \sqcap R) \square (P \sqcap R)$.

We prove this by defining a lemma which says that $revivals_M(P)$ is not equal to $revivals_M(Q)$ if and only if there is a revival r such that $r \in revivals_M(P)$ and $r \notin revivals_M(Q)$ or $r \notin revivals_M(P)$ and $r \in revivals_M(Q)$. It is proven as

```
lemma noteq_revivals:
"(revivals(P) M ~= revivals(Q) M) = ((EX r.  (r :r revivals P
M & r ~:r revivals Q M)) | (EX r.  (r ~:r revivals P M & r :r
revivals Q M)))"
```

We have verified that $(P \Box Q) \sqcap R \neq_{\mathcal{R}} (P \sqcap R) \Box (P \sqcap R)$ in our implementation by giving witness such that $(\langle\rangle, \{b\}, a) \in \mathit{revivals}_M(P \sqcap R) \Box (P \sqcap R)$ and $(\langle\rangle, \{b\}, a) \notin \mathit{revivals}_M((P \Box Q) \sqcap R)$. It is proven in the below lemma

```
lemma InternalDistributesOverExternalCounterExample:    "a~=b -->
~( (a -> STOP [+] b -> STOP) |~| STOP =R[M,M] (a -> STOP |~|
STOP) [+] (b -> STOP |~| STOP) ) "
```

In [Ros07], two weaker versions of ($\Box$-$\sqcap$-dist) are given. For any set of events A, and if $P(x)$ and $Q(x)$ are processes for each $x \in A$, then the internal choice operator is distributive with respect to the prefix choice operator *prefix-choice*-dist is given as

$$?\,x : A \to (P(x) \sqcap Q(x)) =_{\mathcal{R}} (?\,x : A \to P(x)) \sqcap (?\,x : A \to Q(x))$$

This says that an event in A followed by an internal choice is the same as the internal choice made before the event. The stable revivals model has no memory whether the events in A has happened before or after the internal choice was made. A generalised version of distribution is given below

$((?\,x : A \to P(x)) \sqcap (?\,x : B- > Q(x))) =_{\mathcal{R}}$
$?\,x : A \to (\mathtt{IF}\,(x \in A \cap B)\,\mathtt{THEN}\ P(x) \sqcap Q(x)\,\mathtt{ELSE}\ (\mathtt{IF}\ x \in A\ \mathtt{THEN}P(x)\ \mathtt{ELSE}\ Q(x)))$
$\sqcap$
$?\,x : B \to (\mathtt{IF}\,(x \in A \cap B)\ \mathtt{THEN}\ P(x) \sqcap Q(x)\,\mathtt{ELSE}\ (\mathtt{IF}\ x \in A\ \mathtt{THEN}P(x)\ \mathtt{ELSE}\ Q(x)))$

and we verified it in the below lemma.

```
lemma ModifiedInternalDistributesOverExternal_Generised:
"((( ALL a.  P a :  procR ) & ( ALL a.  Q a :  procR )) ==>
((? a:A -> P a) |~| (? a:B -> Q a)) =R[M,M]
((? a:A -> (IF (a : A & a : B) THEN P a |~| Q a ELSE IF (a :
A) THEN P a ELSE Q a) ) |~| (? a:B -> (IF (a : A & a : B) THEN
P a |~| Q a ELSE IF (a : A) THEN P a ELSE Q a))))"
```

7.1.2 Step laws

Step laws are the core of algebraic laws which holds in all the standard models of CSP. It helps us to calculate the first actions of the processes and also plays an important role in the normalisation of a process. The step laws for some operators are given in Figure 7.3.

We have found that the step laws for Stop, the external choice operator, and the renaming operator fail. We modify the semantics for the deadlock component of the prefix choice operator and the renaming operator and prove the step laws.

7.1.2.1 The step law for Stop and the external choice operator

The step laws [Ros98] for `STOP` fails. The step laws of `STOP` is given below:

"cspR_step"

$$\text{STOP} =_{\mathcal{R}} ?\,x : \emptyset \to P(x) \qquad \text{(stop-step)}$$

$$a \to P =_{\mathcal{R}} ?\,x : \{a\} \to P \qquad \text{(prefix-step)}$$

$$\begin{aligned}&(?\,x : A \to P(x)) \,\square\, (?\,x : B \to Q(x))\\ &=_{\mathcal{R}} ?\,x : (A \cup B) \to (\texttt{IF}\,(x \in A \cap B)\,\texttt{THEN}\; P(x) \sqcap Q(x)\\ &\qquad\qquad \texttt{ELSE IF}\,(x \in A)\,\texttt{THEN}\; P(x)\,\texttt{ELSE}\; Q(x)) \qquad (\square\text{-step})\end{aligned}$$

$$\begin{aligned}&(?\,x : A \to P(x))[[r]]\\ &=_{\mathcal{R}} ?\,x : \{x \mid \exists\, a \in A.\,(a,x) \in r\} \to\\ &\quad (!\; a : \{a \in A \mid (a,x) \in r\} \bullet (P(a)[[r]])) \qquad ([[r]]\text{-step})\end{aligned}$$

$$(?\,x : A \to P(x)) \,\mathbin{;}\, Q =_{\mathcal{R}} ?\,x : A \to (P(x) \,\mathbin{;}\, Q) \qquad (\mathbin{;}\text{-step})$$

$$(?\,x : A \to P(x)) \lfloor (n+1) =_{\mathcal{R}} ?\,x : A \to (P(x) \lfloor n) \qquad (\lfloor\text{-step})$$

Figure 7.3: CSP Step laws

$$\text{STOP} =_{\mathcal{R}} ?\,x : \emptyset \to P(x) \qquad \texttt{(stop-step)}.$$

It fails because we have $deadlocks_M(STOP)\ M = \{\langle\rangle\}$, but we do not have $deadlocks_M(? : \{\} \to Qf)M = \{\}$. We have also found a counter example for the step law of the external choice operator if we use the old definition. It is proved in the below lemma

```
lemma cspR_Ext_choice_step_counter_example:
"~(((? x:{a} -> STOP) [+] (? x:{} -> STOP)) M =R (? x:({a} Un
{}) -> (IF (x : {a} & x : {}) THEN (STOP |~| STOP) ELSE IF (x :
{a}) THEN STOP ELSE STOP)) M)"
```

By modifying the semantic function of deadlocks as

$$deadlocks_M(?\ x : A \to P(x)) = \{\langle x\rangle \frown t' \mid t' \in deadlocks_M(P(x)), x \in A\} \cup \{(<> \mid A = \{\}\}.$$

We have verified all the laws given in Figure 7.3. In Figure 7.3, we left out the condition `P : procR` intentionally. For example, the step law for the sequential operator

$$(?\,x : A \to P(x)) \,\mathbin{;}\, Q =_{\mathcal{R}} ?\,x : A \to (P(x) \,\mathbin{;}\, Q)$$

is proved as

$$Q : procR\ \&\ \forall\ a.\ Pf\ a\ :\ procR\ ==>\ (?\,x : A \to P(x)) \,\mathbin{;}\, Q =_{\mathcal{R}} ?\,x : A \to (P(x) \,\mathbin{;}\, Q)$$

In CSP-Prover, it is proved as

```
lemma cspR_Seq_compo_step:
 "(( ALL a.  Pf a :  procR & Q : procR) ==>
 (?  :X -> Pf) ;; Q =R[M,M] ? x:  X -> (Pf x ;;Q ))"
```

7.1.2.2 The step law for the renaming operator

We have also found that the step laws for the renaming operator fails with new semantic definition of $deadlocks_M(?x : A \rightarrow p(x))$. However the old semantic definition of $deadlocks_M(?x : A \rightarrow p(x))$ is wrong and hence we correct this problem by modifying the semantics of deadlock of the renaming operator.

The below is a counter example where $\Sigma = \{a, b\}$

$$?x : a \rightarrow Skip[[(b, b)]] = ?x : \{\} \rightarrow Skip$$

The left hand side process is not a deadlock process as $deadlocks_M(?x : a \rightarrow Skip[[(b, b)]]) = \{\}$, but the right hand side process is a deadlock process as $deadlocks_M(?x : \{\} \rightarrow Skip) = \{\langle\rangle\}$.

The new semantics for the deadlock component of the renaming operator is defined in a way similar to the semantics of the parallel operator: calculating the semantics for the deadlocks component from the failures component.

The new semantics for the renaming operator is

$$deadlock(P[[R]]) \quad = \quad \{ s' \mid \exists s \,.\, sR^*s' \wedge (s, R^{-1}(\Sigma^{\checkmark})) \in failures(P)\}$$

Using the above semantics, we have proven the step law for the renaming operator.

Consider the above example. As the semantics for the deadlocks component does not affect the revivals and traces component, we consider only the revivals component. In the example $R^{-1}(\Sigma^{\checkmark}) = \{b, \checkmark\}$, thus we have $(\langle\rangle, \{b, Tick\}) : failures(?x : \{a\} \rightarrow Skip)$. By definition of $deadlocks_M(P[[R]])$ and $deadlocks_M(?x : A \rightarrow p(x))$, we get $\{\langle\rangle\}$ in both sides for the deadlock component.

7.2 An example for recursive process

In this section, we are going to look at an example of how recursive processes are defined and used in the stable revivals model. Since we have proved the type correctness for a restricted language of CSP_{TP} where renaming relation r is finite, we need to assume each process P is in $procR$. Consider a event set $\Sigma = \{a, b\}$. We consider two processes P and Q. These two are processes defined in terms of themselves.

$$P = a \rightarrow \$P \;\Box\; b \rightarrow \texttt{STOP}$$

$$Q = a \rightarrow \$Q \sqcap b \rightarrow \texttt{STOP}$$

P is a deterministic process which offers the event a indefinitely but terminates by offering the event b at any time. Q is a non-deterministic process which performs the event a indefinitely but terminates by performing the event b and internal non deterministic choice is made at any time.

The denotational semantic of P is given below
$(\{\langle\rangle, \langle b\rangle, \langle a\rangle, \langle a, b\rangle, \langle a, a\rangle, \langle a, a, b\rangle, \ldots\},$
$\{\langle b\rangle, \langle a, b\rangle, \langle a, a, b\rangle, \ldots\},$
$\{(\langle\rangle, \{\}, a), (\langle\rangle, \{\}, b),$
$(\langle a\rangle, \{\}, a), (\langle a\rangle, \{\}, b),$
$(\langle a, a\rangle, \{\}, a), (\langle a, a\rangle, \{\}, b),$
$(\langle a, a, a\rangle, \{\}, a), (\langle a, a, a\rangle, \{\}, b),$

$\vdots$
})

The denotational semantic of Q is given below
$(\{\langle\rangle, \langle b\rangle, \langle a\rangle, \langle a, b\rangle, \langle a, a\rangle, \langle a, a, b\rangle, \ldots\},$
$\{\langle b\rangle, \langle a, b\rangle, \langle a, a, b\rangle, \ldots\},$
$\{(\langle\rangle, \{\}, a), (\langle\rangle, \{\}, b), (\langle\rangle, \{b\}, a), (\langle\rangle, \{a\}, b),$
$(\langle a\rangle, \{\}, a), (\langle a\rangle, \{\}, b), (\langle a\rangle, \{b\}, a), (\langle a\rangle, \{a\}, b),$
$(\langle a, a\rangle, \{\}, a), (\langle a, a\rangle, \{\}, b), (\langle a, a\rangle, \{b\}, a), (\langle a, a\rangle, \{a\}, b),$
$(\langle a, a, a\rangle, \{\}, a), (\langle a, a, a\rangle, \{\}, b), (\langle a, a, a\rangle, \{b\}, a), (\langle a, a, a\rangle, \{a\}, b),$
$\vdots$
})

It is clear that Q is refined by P by the model R $(Q \sqsubseteq_{\mathcal{R}} P)$ as every trace of P is a trace of Q, every deadlock of P is a deadlock of Q, and every revival of P is a revival of Q.

We verify this using our implementation. In CSP-Prover the communication alphabet Σ and the process names Π should be declared before use as in Isabelle. We declare using the keyword `datatype` below:

```
datatype Event  = a | b
datatype PName  = P
datatype QName  = Q
```

Recursive processes in CSP-Prover should be declared along with the functions which will be made as an instance of process name function by overloading later. We declare for the process name $\$P$ using the function `procPfun`.

```
consts
 procPfun ::   "PName ==> (PName, Event) proc"
primrec
 "procPfun P = a -> $P [+] b -> STOP"
```

Processes involving process names are declared as an equation. A new process name is introduced on the left hand side; on the right hand is that a process expression involving process name with dollar symbol attached in prefix is defined as `a -> $P [+] b -> STOP`. The function `procPfun` is defined to be the process name function `PNfun` using `overload` keyword by the following command:

```
defs (overloaded)
set_procPfun_def [simp]:   "PNfun == procPfun"
```

We define the process $\$P$ separately as a constant definition

```
consts
 procP ::   "(PName, Event) proc"
defs
 procP_def :   "procP == $P"
```

Similarly we do for the process name Q and declare `CPO_mode` approach.

```
consts
 procQfun ::  "QName ==> (QName, Event) proc"
primrec
 "procQfun Q = a --> $Q |~| b --> STOP"
defs (overloaded)
 set_procQfun_def [simp]:  "PNfun ==> procQfun"
consts
 procQ ::  "(QName, Event) proc"
defs
 procQ_def :  "procQ == $Q
defs
 FPMode_def [simp]:  "FPmode ==CPOmode"
```

We relate processes to compare P and Q by mapping process name P to process $\$Q$ by the following command:

```
consts
 map_Pproc_to_Qproc ::  "PName ==> (QName, Event) proc"
primrec
 "map_Pproc_to_Qproc P = $Q"
```

Now we define the refinement relation $Q \sqsubseteq_{\mathcal{R}} P$ by the following lemma:

```
lemma refinment:  "[| ALL i.  (procPfun i) :  procR ;
ALL x .  (procPfun x) << map_Pproc_to_Qproc :  procR
ALL i.  map_Pproc_to_Qproc i:  procR;
ALL i.  (procQfun i) :  procR ; |] ==> procQ <=R procP"
```

We need to assume that functions `procPfun` and `procQfun` are in set *procR*. We also have to assume process map function `map_Pproc_to_Qproc` and its substitution `(procPfun x) << map_Pproc_to_Qproc` are in `procR`. After unfolding the definition of `procQ` and `procP`, we get the following subgoal:

```
[| ALL x::PName.  (procPfun x) << map_Pproc_to_Qproc :  procR;
 ALL i::PName.  map_Pproc_to_Qproc i :  procR;
 ALL i::PName.  procPfun i :  procR; ALL i::QName.  procQfun i :
procR|]                               ==> $Q <=R $P
```

Since we are mapping process name P to process $\$Q$ and process $\$P$ appear on the right hand of refinement relation, we convert `$Q <=R $P` into `$Q <=R map_Pproc_to_Qproc P` by passing process map function as an argument to introduction rule `cspR_fp_induct_cpo_right`. By the definition of `map_Pproc_to_Qproc P = $Q`, it trivially follows that `$Q <=R $Q`.

Now we need to prove the following important subgoal:

```
!!p::PName.  [| ALL i::PName.  procPfun i :  procR; ALL i::QName.
procQfun i :  procR;
 ALL i::PName.  map_Pproc_to_Qproc i :  procR;
 ALL x::PName.  (procPfun x) << map_Pproc_to_Qproc :  procR |]
 ==> map_Pproc_to_Qproc p <=R (procPfun p) << map_Pproc_to_Qproc
```

This we prove the goal by structural induction on p as `PNames` is defined by keyword `primrec`. After applying the command `apply (induct_tac p)` yields the following:

```
!!p::PName.  [|ALL i::PName.  procPfun i :  procR; ALL i::QName.
procQfun i :  procR;
ALL i::PName.  map_Pproc_to_Qproc i :  procR;
 ALL x::PName.  (procPfun x) << map_Pproc_to_Qproc :  procR |]
 ==> map_Pproc_to_Qproc P <=R (procPfun P) << map_Pproc_to_Qproc
```

We know that `map_Pproc_to_Qproc P` is equal to $\$Q$. By definition of `procPfun`, it follows that `(procPfun P) = a -> $P [+] b -> STOP`.

We have also declared `map_Pproc_to_Qproc P = $Q`. By definition of <<, we get
`a -> $P [+] b -> STOP << map_Pproc_to_Qproc = a -> $P [+] b -> STOP`.
After applying the command `apply (simp)`, we get the following subgoal:

```
[| ALL i::PName.  procPfun i :  procR; ALL i::QName.  procQfun i
:  procR;
 ALL i::PName.  map_Pproc_to_Qproc i :  procR;
 ALL x::PName.  (procPfun x) << map_Pproc_to_Qproc :  procR |]
 ==> $Q <=R a -> $Q [+] b -> STOP
```

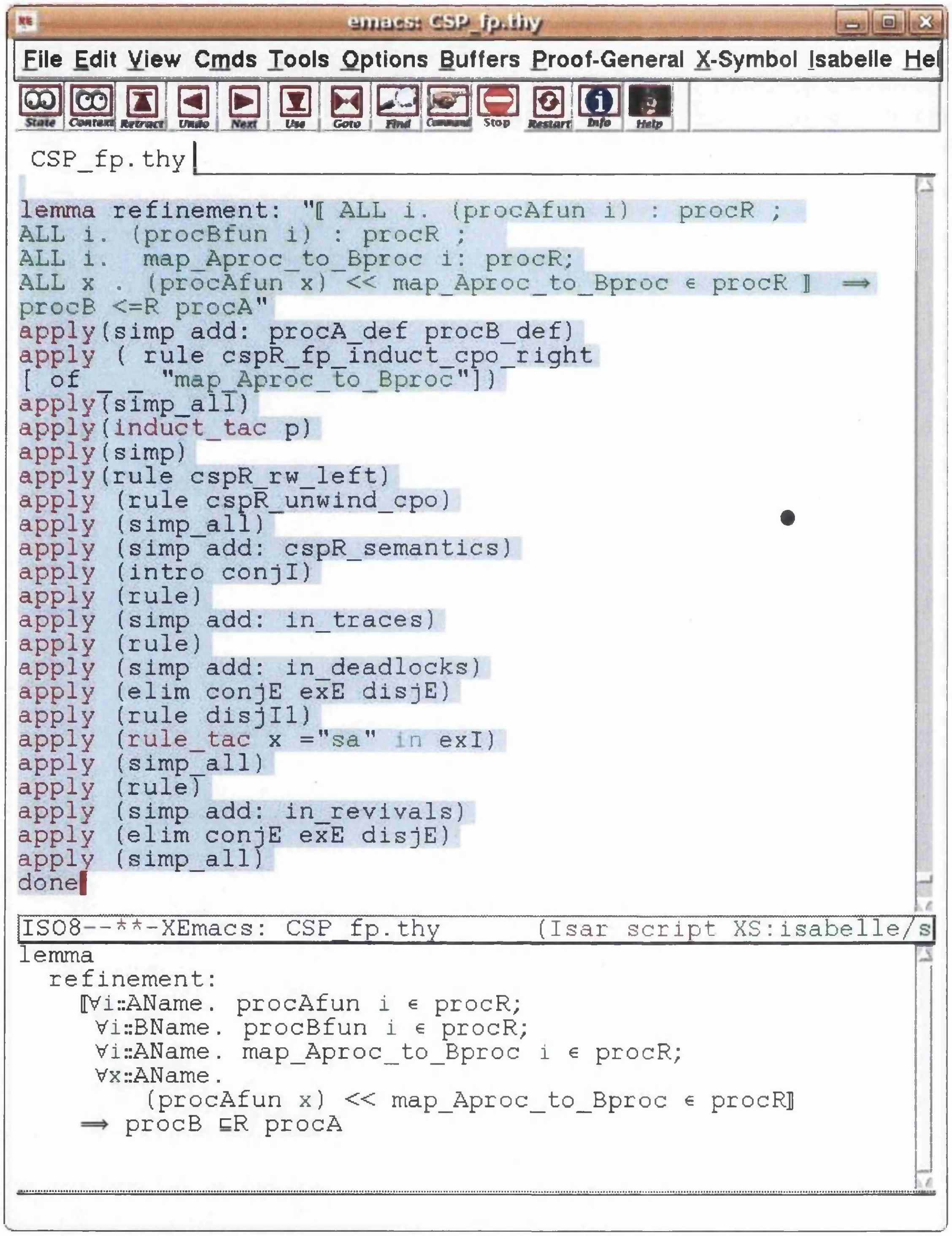
After applying unwind laws and simplification using the below commands:

```
apply (rule cspR_rw_left)
apply (rule cspR_unwind_cpo)
apply (simp_all)
apply (simp)
```

We obtain the following goal:

```
[| ALL i::PName.  procPfun i :  procR;
 ALL i::QName.  procQfun i :  procR;
 ALL i::PName.  map_Pproc_to_Qproc i :  procR;
 ALL x::PName.  (procPfun x) << map_Pproc_to_Qproc :  procR |]
 ==> a -> Q |~| b -> STOP <=R a -> $Q [+] b -> STOP
```

Then by applying the definition of semantic function, we can prove the above goal easily. Figure 7.4 shows screen-shot of the proof in CSP-Prover.


emacs: CSP_fp.thy
File Edit View Cmds Tools Options Buffers Proof-General X-Symbol Isabelle Hel
State Context Retract Undo Next Use Goto Find Command Stop Restart Info Help
CSP_fp.thy

```
lemma refinement: "⟦ ALL i. (procAfun i) : procR ;
ALL i. (procBfun i) : procR ;
ALL i.  map_Aproc_to_Bproc i: procR;
ALL x . (procAfun x) << map_Aproc_to_Bproc ∈ procR ⟧  ⟹
procB <=R procA"
apply(simp add: procA_def procB_def)
apply ( rule cspR_fp_induct_cpo_right
[ of _ _ "map_Aproc_to_Bproc"])
apply(simp_all)
apply(induct_tac p)
apply(simp)
apply(rule cspR_rw_left)
apply (rule cspR_unwind_cpo)
apply (simp_all)
apply (simp add: cspR_semantics)
apply (intro conjI)
apply (rule)
apply (simp add: in_traces)
apply (rule)
apply (simp add: in_deadlocks)
apply (elim conjE exE disjE)
apply (rule disjI1)
apply (rule_tac x ="sa" in exI)
apply (simp_all)
apply (rule)
apply (simp add: in_revivals)
apply (elim conjE exE disjE)
apply (simp_all)
done
ISO8--**-XEmacs: CSP_fp.thy        (Isar script XS:isabelle/s
lemma
  refinement:
    ⟦∀i::AName. procAfun i ∈ procR;
     ∀i::BName. procBfun i ∈ procR;
     ∀i::AName. map_Aproc_to_Bproc i ∈ procR;
     ∀x::AName.
        (procAfun x) << map_Aproc_to_Bproc ∈ procR⟧
    ⟹ procB ⊑R procA
```

Figure 7.4: Refinement relation having recursive process

Chapter 8

An Improved Model

Contents

In this chapter, we present an improved stable revivals model given by Y. Isobe. The improved model is same as the stable revivals model except that one of the healthiness conditions of the stable revivals model is replaced by a new healthiness condition. In this chapter, firstly, we explain the need for an improvement to Roscoe's stable revivals model. Secondly, we define the improved stable revivals model. Finally we conclude the chapter by explaining the implementation of the improved model in CSP-Prover.

8.1 Need for an Improved model

Consider again the counter example given in Section 6.1.1 to show another problem:

$$C = (\{\langle\rangle, \langle a\rangle, \langle b\rangle\},\ \emptyset,\ \{(\langle\rangle, X, a),\ (\langle\rangle, X, b) \mid X \in \mathcal{P}_{fin}(\mathcal{N})\})$$

C fulfils all the healthiness conditions of the model $\mathcal{R}$. However, there is no process which represent the denotation C. This example is also a counter example for the full abstraction property when Σ is infinite.

In [Ros07], the full abstraction property is proved assuming Σ is finite. This assumption has been made to prove the full abstraction property of the stable revivals model. In CSP$_{\mathrm{TP}}$, Σ can be infinite. We have restricted the language of CSP$_{\mathrm{TP}}$ to use the stable revivals model proposed in [Ros07] and proved all the properties mentioned in the previous chapters and verified some algebraic laws. We have used the language CSP$_{\mathrm{TP}}$ by restricting that the renaming relation to be finite in the renaming operator. This restriction has complicated the proofs. As we have to assume that all processes satisfy this restriction. Consider a small example which explains the problem. We have proved that the external choice operator is idempotent by assuming P is in the restricted language of CSP$_{\mathrm{TP}}$ as given below:

```
lemma External_Idem:   " P : procR ==> (P [+] P) =R P "
```

Similarly, we have proved the type correctness of the semantic function of the stable revivals model and the semantic function of the stable revivals model is continuous by assuming this restriction as follows:

```
lemma continuous_semRf:   "P : procR ==> continuous [[P]]Rf"
lemma proc_domR_lm[simp]:   "P : procR ==>
  (traces(P) (fstR o M), deadlocks(P) M, revivals(P) M) : domR"
```

All the proofs which used these properties should have this restriction. Hence it has complicated the implementation. We need another improved model which removes the restriction and also satisfying all the properties of the stable revivals model.

8.2 Improved model

In the previous section, we explained the need for an improved model which removes the restriction that renaming relation is finite and also satisfies all the properties of the stable revivals model. In this section, we define the extension of the stable revivals model and explain its relation with the old one.

We are unable to prove the type correctness of the renaming operator as it does not satisfy **R3** without any restriction. Y. Isobe proposed a condition **R3′** replacing old **R3**. The new healthiness condition is given below:

R3′ $((s, X, a) \in R \wedge Y \subseteq \Sigma \wedge (\forall b.b \in Y \Longrightarrow (s, X, b) \notin R)) \Longrightarrow (s, X \cup Y, a) \in R.$

This says that all events which would not happen in the next step will be refused.

This condition is similar to the following healthiness condition in the stable failures model. However we do not need to use the traces component to find the next acceptance event.

F3. $((s, X) \in \mathit{failures}(P) \wedge \forall a \in Y . s \frown \langle a \rangle \notin \mathit{traces}(P)) \Longrightarrow (s, X \cup Y) \in \mathit{failures}(P).$

This says that all the events which does not happen after the traces s, should be refused.

We can also observe that $R3'$ implies $R3$. This gives rise to two different semantic domains. They are $\mathit{dom}(\mathcal{R})$ which fulfils **T1, D1**, **R1**, **R2**, **R3**, and **RSS05** and $\mathit{dom}(\mathcal{R}')$ which fulfils **T1, D1**, **R1**, **R2**, **R3′**, and **RSS05**. Clearly, $\mathit{dom}(\mathcal{R}') \subseteq \mathit{dom}(\mathcal{R})$ as **R3′** implies **R3**. We also verified in Isabelle by the following lemma:

```
lemma R3primeImpliesR3:
"( ALL s X a Y. (s, X,a) :r (revivals P M) & Y<= Evset &
 (ALL b.  b :  Y --> (s, X,b) ~:r (revivals P M))
 --> (s, X Un Y,a) :r (revivals P M) ) ==>
(ALL s X b c.
(s, X, b) :r (revivals P M) & Tick ~:  X & noTick s & c ~= Tick
-->
(s, X, c) :r (revivals P M) | (s, insert c X, b) :r (revivals P
M))"
apply (intro allI impI)
apply (elim conjE exE disjE)
apply (drule_tac x="s" in spec)
apply (drule_tac x="X" in spec)
apply (drule_tac x="b" in spec)
apply (drule_tac x="{ c }" in spec)
apply (auto simp add:  Evset_def)
done
```

We now give the proof that *revivals*(P) satisfies **R3′**.

Lemma 8.1: Assuming that P satisfies **R3′**, we prove that $(P[\![R]\!])$ satisfies **R3′**

Proof. Suppose P satisfies **R3′** which says

$$\begin{aligned}&((s, R^{-1}(X), a) \in \mathit{Revivals}(P) \wedge R^{-1}(Y) \subseteq \Sigma \wedge \\ &\forall\, b.\ b \in R^{-1}(Y) \Longrightarrow (s, R^{-1}(X), b) \notin \mathit{Revivals}(P)) \\ &\quad \Longrightarrow (s, R^{-1}(X) \cup R^{-1}(Y), a) \in \mathit{Revivals}(P).\end{aligned}$$

We prove that $(P[\![R]\!])$ satisfies **R3′** that is

$$\begin{aligned}&(s', X, a') \in \mathit{Revivals}(P[\![R]\!]) \wedge Y \subseteq \Sigma \wedge \\ &\forall\, b'.\ b' \in Y \Longrightarrow (s', X, b') \notin \mathit{Revivals}(P[\![R]\!]) \\ &\quad \Longrightarrow (s', X \cup Y, a') \in \mathit{Revivals}(P[\![R]\!]).\end{aligned}$$

Suppose $(s', X, a') \in \mathit{Revivals}(P[\![R]\!])$, $Y \subseteq \Sigma$, and $\forall\, b'.\ b' \in Y \Longrightarrow (s', X, b') \notin \mathit{Revivals}(P[\![R]\!])$. By definition of $\mathit{Revivals}(P[\![R]\!])$, for some s and a,

$$(s\, R\, s'), (a, a') \in R \text{ and } (s, R^{-1}(X), a) \in \mathit{Revivals}(P).$$

At first, we prove that $\forall\, b.\ b \in R^{-1}(Y) \Longrightarrow (s, R^{-1}(X), b) \notin \mathit{Revivals}(P)$. Let $b \in R^{-1}(Y)$. Since $R^{-1}(Y) = \{b \mid \exists\, b' \in Y.\ (b = b' = \checkmark \vee (b, b') \in R)\}$, we consider the following two cases.

- case i: $b = \checkmark$
  By the definition of R^{-1}, $\checkmark \in Y$. But we know that $Y \subseteq \Sigma$. Hence contradiction.

- case ii: $b \neq \checkmark$
  By the definition of R^{-1}, there exists $b' \in Y$ such that $(b, b') \in R$. From $b' \in Y$ and the

assumption ($\forall\, b'.\ b' \in Y \Longrightarrow (s', X, b') \notin \mathit{Revivals}(P[\![R]\!])$), we have

$$(s', X, b') \notin \mathit{Revivals}(P[\![R]\!]).$$

By definition of $\mathit{Revivals}(P[\![R]\!])$, for all s'' and b'',

$$\mathit{not}\ (s'' R s') \vee (\ b'', b') \notin R \vee (s'', R^{-1}(X), b'') \notin \mathit{Revivals}(P).$$

Now set $s'' = s$ and $b'' = b$. In this case, we already have $(s\ R\ s')$ and $(b, b') \in R$. Hence we necessarily have $(s, R^{-1}(X), b) \notin \mathit{Revivals}(P)$.

Consequently, $\forall\, b.\ b \in R^{-1}(Y) \Longrightarrow (s, R^{-1}(X), b) \notin \mathit{Revivals}(P)$.

Next, since P satisfies **R3**$'$, we have $(s, R^{-1}(X) \cup R^{-1}(Y), a) \in \mathit{Revivals}(P)$ because $(s, R^{-1}(X), a) \in \mathit{Revivals}(P)$ and $R^{-1}(Y) \subseteq \Sigma$. Finally, by the definition of $\mathit{Revivals}(P[\![R]\!])$, we have $(s', X \cup Y, a') \in \mathit{Revivals}(P[\![R]\!])$. Hence it completes the proof. □

This proof enables us to prove the type correctness without any restriction. This is the only modification which needs to be done. In the next section, we briefly discuss the implementation of this improved model.

8.3 Implementation of Improved Model

We have implemented the improved model in CSP-Prover similar to the approach we followed in the previous chapters. All of the properties which we have verified in the previous chapters follow directly. Encoding of most of the implementation is similar to the previous implementation. Hence we just describe the results which require modification. We do not need to do any modification for the deadlock and traces components or semantic definitions of any component.

Like the previous implementation, we encode the new healthiness condition **R3**$'$ in the domain of the revivals component. It is defined as follows:

```
consts
 HC_R3 ::  " 'a revival set => bool"
defs
 HC_R3_def :  " HC_R3 F == (ALL s X a Y. ( (s, X,a) :  F &
noTick s & Y <= Evset & (ALL b.  b :  Y --> (s, X, b) ~:  F) -->
(s, X Un Y,a) :  F))"
```

We create a new type for revivals component using `typedef` command in theory `set_R` as follows:

```
typedef 'a setR = "{ R ::('a revival set ) .
               HC_RT(R) & HC_RF(R) & HC_R2(R) & HC_R3(R)}"
 apply (rule_tac x ="{}" in exI)
 by (simp add:  HC_RT_def HC_RF_def HC_R2_def HC_R3_def)
```

This creates a set of all revivals and we finish the proof by showing that the set is non-empty by giving the empty set as a witness. We make the newly created type to be instance of the axiomatic type class `cpo_bot` in theory `set_R_cpo`.

```
instance setR ::  (type) cpo_bot
apply (intro_classes)
...
```

We implement the domain of the stable revivals model in theory Domain. This does not require any modification as all the changes have been done in theory set_R itself. We prove the type correctness. for the revivals component in the theory CSP_R_revivals. We prove the type correctness of the renaming operator for revivals component and the domain of the extended model in the below lemmas:

```
lemma RENAMING_setR: "{rr.  EX sa t X a aa .  rr = (t,X,aa) &
(sa,[[r]]inv X,a) :r P & (sa [[r]]* t) &
(a,aa) :  EventPairSet (r::  ('a × 'a) set) } :  setR"
```

```
lemma RENAMING_domR:
"(traces(P) (fstR o M), deadlocks(P) M, revivals(P) M ) : domR
 ==> (traces(P [[r]]) (fstR o M), deadlocks(P [[r]]) M,
revivals(P [[r]]) M) : domR"
```

In the previous version, we have implemented both the lemmas as follows:

```
lemma RENAMING_setR: "finite r -->
 {rr.  EX sa t X a aa .  rr=(t,X,aa) & (sa,[[r]]inv X,a) :r P &
(sa [[r]]* t) &
 (a,aa) :  EventPairSet (r::  ('a × 'a) set) } :  setR"

lemma RENAMING_domR: "[| finite r ;
(traces(P) (fstR o M), deadlocks(P) M, revivals(P) M ): domR |]
 ==> (traces(P [[r]]) (fstR o M), deadlocks(P [[r]]) M,
revivals(P [[r]]) M) : domR"
```

The type correctness and continuity of semantic function of the improved model is proved in the below.

```
lemma proc_domR_lm[simp]:
"(traces(P) (fstR o M), deadlocks(P) M, revivals(P) M) : domR"

lemma continuous_semRf:  "continuous [[P]]Rf"
```

In the previous implementation, we have proved the above lemmas as

```
lemma proc_domR_lm[simp]:
"P:procR--> (traces(P) (fstR o M), deadlocks(P) M, revivals(P)
M):domR"

lemma continuous_semRf:  "P : procR ==> continuous [[P]]Rf"
```

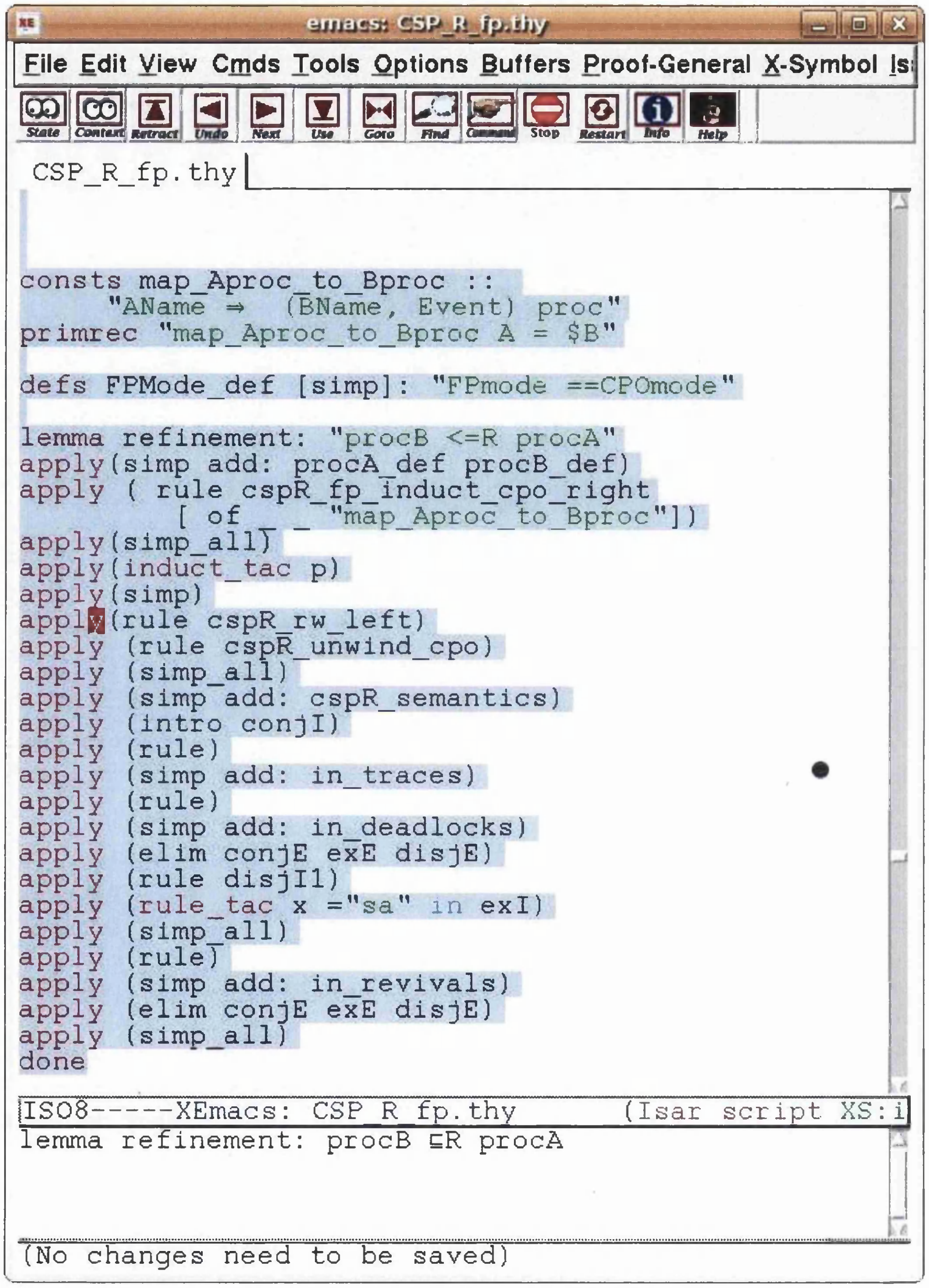
These are the major benefits that we get in the improved model. This also reduces nearly 1000 lines of code. The code compiles faster then the previous implementation. Consider the example of recursive process which we have proved in the previous chapter. In the new improved model, the lemma is given as

```
lemma refinement:  "procB <=R procA"
```

In the previous implementation, the lemma looks like

```
lemma refinment:  "[| ALL i.  (procPfun i) :  procR ;
ALL i.  (procQfun i) :  procR ;
ALL i.  map_Pproc_to_Qproc i:  procR;
ALL x .  (procPfun x) << map_Pproc_to_Qproc :  procR |]
 ==> procQ <=R procP"
```

Figure 8.1 shows the screen shot of the proof.


emacs: CSP_R_fp.thy
File Edit View Cmds Tools Options Buffers Proof-General X-Symbol Is
State Context Retract Undo Next Use Goto Find Command Stop Restart Info Help
CSP_R_fp.thy

```
consts map_Aproc_to_Bproc ::
     "AName ⇒  (BName, Event) proc"
primrec "map_Aproc_to_Bproc A = $B"

defs FPMode_def [simp]: "FPmode ==CPOmode"

lemma refinement: "procB <=R procA"
apply(simp add: procA_def procB_def)
apply ( rule cspR_fp_induct_cpo_right
         [ of _ _ "map_Aproc_to_Bproc"])
apply(simp_all)
apply(induct_tac p)
apply(simp)
apply(rule cspR_rw_left)
apply (rule cspR_unwind_cpo)
apply (simp_all)
apply (simp add: cspR_semantics)
apply (intro conjI)
apply (rule)
apply (simp add: in_traces)
apply (rule)
apply (simp add: in_deadlocks)
apply (elim conjE exE disjE)
apply (rule disjI1)
apply (rule_tac x ="sa" in exI)
apply (simp_all)
apply (rule)
apply (simp add: in_revivals)
apply (elim conjE exE disjE)
apply (simp_all)
done
ISO8-----XEmacs: CSP_R_fp.thy        (Isar script XS:i
lemma refinement: procB ⊑R procA
(No changes need to be saved)
```

Figure 8.1: Refinement relation in the new extension model

Chapter 9

Conclusion

Contents

In this chapter, we summarise the work done in this project and outline future work. As noted in the introduction chapter, our main reason for implementing the model was to provide theorem proving support for the stable revivals model, and to verify the type correctness and continuity of the semantic function of the stable revivals model. We summarise the work done in this project and conclude the chapter with a discussion of some of the properties still to be implemented.

9.1 Summary

In this project, we studied the newly developed model, the stable revivals model $\mathcal{R}$. We presented a faithful and running implementation of the stable revivals model in the proof tool CSP-Prover. This requires certain changes with respect to Roscoe's original proposed model in the implementation.

- Inclusion of healthiness condition **RRS05** in the definition of the domain.
- Change of a semantic clause in the definition of $deadlocks_M(?\,x : A \rightarrow (P(x)))$.
- Modification of the semantic function for the deadlock component of the renaming operator.
- Restriction of the language CSP_{TP} to finite renaming relations.

We have implemented two versions of the stable revivals model:

- Roscoe's original model with the changes as described above, and
- An improvement of this model as described in Chapter 8.

Currently this initial phase of the work is concentrating on encoding the semantic function of the stable revivals model and proving the important properties like the type correctness and continuity of semantic functions. Though the proof ideas implemented in the stable revivals models is similar to the previous models implemented in CSP-Prover, the proof does not follow directly in most of the cases as we have to restrict the language CSP_{TP}.

In this project, we have also demonstrated the potential of our implementation:

- By proving a number of algebraic laws. We have mainly proved the step laws of the some selected operators.
- A refinement example including recursive processes.

The tool support for the stable revivals model will encourage those working in industry to use this implementation for practical applications. We have also presented an amendment to Roscoe's model which allows removing restriction on the language CSP_{TP}. However the theoretical implication of the proposal are beyond the scope of this project.

9.2 Future Work

In this section, we outline and give a direction to the future work of this project.

- This work has provided the basic infrastructure for the verification of practical applications. To successfully demonstrate our implementation in practical applications, a major case study of the implementation is needed. A good example would be on-line shopping example given in [RRS06]. We have seen that responsiveness and stuckness property can be captured in this model; hence it is interesting to verify these properties.
- We have proved some algebraic laws in Chapter 7. The stable failures model in CSP-Prover has more than 80 laws which allows to prove a complete axiomatic semantics for the CSP stable failures model. It is interesting to do the same for the stable revivals model. Proving more algebraic laws will be helpful to the user of the tool. Currently we can verify the proofs using semantic functions. Following the lines of the previous implementations in CSP-Prover, it is useful to develop tactics to prove refinement and equivalence proofs by semi-automatic methods using the algebraic laws.
- Implementing complete metric based (CMS) approach to reason about recursive processes. This will allows us to find the unique fixed point solution for the recursive equations.

Appendix A

A.1 The Syntax of the core CSP

The below figure gives the syntax of the core CSP language.

$P ::= \mathtt{SKIP}$	%% successful terminating process
$\mid \mathtt{STOP}$	%% deadlock process
$\mid \mathtt{DIV}$	%% divergence
$\mid a \longrightarrow P$	%% action prefix
$\mid ?x : X \longrightarrow P(x)$	%% prefix choice
$\mid P \,\Box\, P$	%% external choice
$\mid P \sqcap P$	%% internal choice
$\mid \bigsqcap S$	%% the generalised internal non-deterministic choice
$\mid \mathtt{IF}\ b\ \mathtt{THEN}\ P\ \mathtt{ELSE}\ P$	%% conditional
$\mid P \,[\![X]\!]\, P$	%% generalized parallel
$\mid P \setminus X$	%% hiding
$\mid P[[R]]$	%% relational renaming
$\mid P \,;\, P$	%% sequential composition
$\mid P \,\triangle\, P$	%% Interrupt operator
$\mid P \rhd P$	%% Time out operator

where $X \subseteq \Sigma$, $b \in Bool$, $a \in \Sigma$, $R \in \mathbb{P}(\Sigma \times \Sigma)$, and S(a set of processes).

A.2 The Semantic functions of the core CSP

The below figures give the semantic function for each individual components in the stable revivals model.

$$\begin{aligned}
revivals(\mathtt{SKIP}) &= \{\} \\
revivals(\mathtt{STOP}) &= \{\} \\
revivals(\mathtt{DIV}) &= \{\} \\
revivals(a \rightarrow P) &= \{(\langle\rangle, X, a) \mid a \notin X\} \\
&\quad \cup \{(\langle a\rangle ^\frown t', X, b) \mid (t', X, b) \in revivals(P)\} \\
revivals(?\, x : A \rightarrow P(x)) &= \{((\langle\rangle, X) \mid A \cap X = \emptyset\} \\
&\quad \cup \{(\langle x\rangle ^\frown t', X, b) \mid (t', X, b) \in revivals(P(x)), x \in A\} \\
revivals(P \sqcap Q) &= revivals(P) \cup revivals(Q) \\
revivals(P \,\Box\, Q) &= \{(\langle\rangle, X, a) \mid (\langle\rangle, X) \in failures^b(P) \cap failures^b(Q) \\
&\quad \wedge (\langle\rangle, X, a) \in revivals(P) \cup revivals(Q)\} \\
&\quad \cup\{(s, X, a) \mid (s, X, a) \in revivals(P) \cup revivals(Q) \wedge s \neq \langle\rangle\} \\
revivals(\textstyle\bigsqcap S) &= \textstyle\bigcup\{revivals(s) \mid s \in S\} \\
revivals(P \,|[\, X \,]|\, Q) &= \{(u, Y \cup Z), a) \mid \\
&\quad \exists\, s, t.(s, Y) \in failures(P) \wedge (t, Z) \in failures(Q) \\
&\quad \wedge\, u \in u \in s \,|[X]|\, t \cap \Sigma^* \\
&\quad \wedge\, Y - (X \cup \{\checkmark\}) = Z - (X \cup \{\checkmark\}) \\
&\quad \wedge\, ((a \in X \wedge (s, Y, a) \in revivals(P) \wedge (t, Z, a) \in revivals(Q)) \\
&\quad\ \ \vee\, a \notin X \wedge (s, Y, a) \in revivals(P) \\
&\quad\ \ \vee\, a \notin X \wedge (t, Z, a) \in revivals(Q)))) \,\}. \\
revivals(\mathtt{IF}\ b\ \mathrm{THEN}\ P\ \mathrm{ELSE}\ Q) &= \text{if } b \text{ evaluates to } \mathit{True} \text{ then } revivals(P) \text{ else } revivals(Q) \\
revivals(P[[R]]) &= \{(s', X, a') \mid \exists\, s, a \,.\, sR^*s' \wedge a\,R\,a' \wedge (s, R^{-1}(X), a) \in revivals(P)\} \\
revivals(P \,\fatsemi\, Q) &= \{(s, X, a) \mid (s, X, a) \in revivals(P)\} \\
&\quad \cup \{(s ^\frown t, X, a) \mid s ^\frown \langle\checkmark\rangle \in traces(P) \wedge (t, X, a) \in revivals(Q)\} \\
revivals(P \,\triangle\, Q) &= \{(s, X, a) \in revivals(P) \mid (\langle\rangle, X) \in failures^b(Q)\} \\
&\quad \cup\{(s, X, a) \mid (s, X) \in failures^b(P) \wedge (\langle\rangle, X, a) \in revivals(Q)\} \\
&\quad \cup\{(s ^\frown t, X, a) \mid s \in traces(P) \cap \Sigma^* \wedge t \neq \langle\rangle \wedge (t, X, a) \in revivals(Q)\} \\
revivals(P \rhd Q) &= revivals(Q) \cup \{(s, X, a) \in revivals(P) \mid s \neq \langle\rangle\}
\end{aligned}$$

where $X \subseteq \Sigma$, $b \in Bool$, $a \in \Sigma$, $R \in \mathbb{P}(\Sigma \times \Sigma)$, and S(set of processes). M is an environment

Let P be denotational represented as $(Tr(P), Dead(P), Rev(P))$, then

$$\begin{aligned}
\text{failures(P)} \quad = \quad & \{(s, X) \mid X \subseteq \Sigma^{\checkmark} \wedge s \in Dead(P)\} \\
& \cup\{(s, X), (s, X \cup \{\checkmark\}) \mid (s, X, a) \in Rev(P)\} \\
& \cup\{(s, X) \mid s ^\frown \langle\langle\rangle\rangle \in Tr(P) \wedge X \subseteq \Sigma\} \\
& \cup\{(s ^\frown \langle\checkmark\rangle, X) \mid s ^\frown \langle\langle\rangle\rangle \in Tr(P) \wedge X \subseteq \Sigma^{\checkmark}\}
\end{aligned}$$

$$\begin{aligned}
failures^b(P) \quad = \quad & \{(s, X) \mid X \subseteq \Sigma \wedge s \in Dead(P)\} \\
& \cup\{(s, X) \mid (s, X, a) \in Rev(P)\}
\end{aligned}$$

The semantic function for the deadlock component is given below:

$$
\begin{aligned}
deadlocks(\mathtt{SKIP}) &= \{\} \\
deadlocks(\mathtt{STOP}) &= \{\langle\rangle\} \\
deadlocks(\mathtt{DIV}) &= \{\} \\
deadlocks(a \rightarrow P) &= \{\langle a\rangle \frown t' \mid t' \in deadlocks(P)\} \\
deadlocks(?\ x : A \rightarrow P(x)) &= \{\langle x\rangle \frown t' \mid t' \in deadlocks(P(x)), x \in A\} \\
deadlocks(P \sqcap Q) &= deadlocks(P) \cup deadlocks(Q) \\
deadlocks(P \mathrel{\Box} Q) &= ((deadlocks(P) \cup deadlocks(Q)) \cap \{s \mid s \neq \langle\rangle\}) \\
&\quad \cup (deadlocks(P) \cap deadlocks(Q)) \\
deadlocks(\textstyle\bigsqcap S) &= \textstyle\bigcup\{deadlocks(s) \mid s \in S\} \\
deadlocks(P \mathbin{|[X]|} Q) &= \{u \mid (s, Y) \in failures(P), (t, Z) \in failures(Q)\ . \\
&\quad Y - (X \cup \{\checkmark\}) = Z - (X \cup \{\checkmark\}) \\
&\quad \wedge u \in s \mathbin{|[X]|} t \\
&\quad \wedge \Sigma^{\checkmark} = Y \cup Z\} \\
deadlocks(\mathtt{IF}\ b\ \mathtt{THEN}\ P\ \mathtt{ELSE}\ Q) &= \text{if } b \text{ evaluates to } True \text{ then } deadlocks(P) \text{ else } deadlocks(Q) \\
deadlocks(P \setminus X) &= \{t \setminus X \mid t \in deadlocks(P)\} \\
deadlocks(P[[R]]) &= \{t \mid \exists\, t' \in deadlocks(P).\ (t', t) \in [[R]]^*\} \\
deadlocks(P \mathbin{;} Q) &= deadlocks(P) \cup \{s \frown t \mid s \frown \langle\checkmark\rangle \in traces(P), t \in deadlocks(Q)\} \\
deadlocs(P \mathbin{\triangle} Q) &= \{s \frown t \mid s \in traces(P) \cap \Sigma^* \wedge t \in deadlocks(Q)\} \\
deadlocs(P \rhd Q) &= deadlocks(Q) \cup \{s \in deadlocks(P) \mid s \neq \langle\rangle\}
\end{aligned}
$$

where $X \subseteq \Sigma$, $b \in Bool$, $a \in \Sigma$, $R \in \mathbb{P}(\Sigma \times \Sigma)$, and S(set of processes).

The semantic function for the traces component is given below:

$$
\begin{aligned}
traces(\mathtt{SKIP}) &= \{\langle\rangle, \langle\checkmark\rangle\} \\
traces(\mathtt{STOP}) &= \{\langle\rangle\} \\
traces(\mathtt{DIV}) &= \{\langle\rangle\} \\
traces(a \rightarrow P) &= \{\langle\rangle\} \cup \{\langle a\rangle \frown t' \mid t' \in traces(P)\} \\
traces(?\ x : A \rightarrow P(x)) &= \{\langle\rangle\} \cup \{\langle x\rangle \frown t' \mid t' \in traces(P(x)), x \in A\} \\
traces(P \mathrel{\Box} Q) &= traces(P) \cup traces(Q) \\
traces(P \sqcap Q) &= traces(P) \cup traces(Q) \\
traces(\textstyle\bigsqcap S) &= \textstyle\bigcup\{traces(s) \mid s \in S\} \cup \{\langle\rangle\} \\
traces(\mathtt{IF}\ b\ \mathtt{THEN}\ P\ \mathtt{ELSE}\ Q) &= \text{if } b \text{ then } traces(P) \text{ else } traces(Q) \\
traces(P \mathbin{|[X]|} Q) &= \{t_1 \mathbin{|[X]|} t_2 \mid t_1 \in traces(P),\ t_2 \in traces(Q)\} \\
traces(P \setminus X) &= \{t \setminus X \mid t \in traces(P)\} \\
traces(P[[R]]) &= \{t \mid \exists\, t' \in traces(P).\ (t', t) \in [[r]]^*\} \\
traces(P \mathbin{;} Q) &= (traces(P) \cap \Sigma^*) \\
&\quad \cup \{t_1 \frown t_2 \mid t_1 \frown \langle\checkmark\rangle \in traces(P),\ t_2 \in traces(Q)\} \\
traces(P \mathbin{\triangle} Q) &= traces(P) \cup \{s \frown t \mid s \in traces(P) \cap \Sigma^*, t \in traces(Q)\} \\
traces(P \rhd Q) &= traces(P) \cup traces(Q)
\end{aligned}
$$

where $X \subseteq \Sigma$, $b \in Bool$, $a \in \Sigma$, $R \in \mathbb{P}(\Sigma \times \Sigma)$, and S(set of processes).

A.3 Machine Verifiable Responsive

To verify Q RESPONDSTOLIVE P on J, we have to check for every trace s, whenever $(s, J) \in failures(P \mathbin{|[J]|} Q)$ is true, $(s, J) \in failures(P)$ is also true.

Let $LQ \equiv (Q \mathbin{|[\Sigma - J]|} CHAOS_{(\Sigma - J)}) \setminus (\Sigma - J)$ be a lazy abstraction of Q with repect to $\Sigma - J$. This removes events which are not in J from the alphabets of Q. LQ behaves like Q except that whenever Q can perform an event from $\Sigma - J$, LQ has the choice of either not doing the event or making the

event invisible.
Let R_J be a relation such that it maps each element a of J to both itself and a new element $NEW \in \Sigma$. Let $NEWSTOP$ be a process such that it can perform any event in Σ, but when it performs NEW, then it deadlocks.

$$NEWSTOP \equiv (NEW \rightarrow \text{STOP}) \,\Box\, (x : (\Sigma - \{NEW\} \rightarrow NEWSTOP)$$

Let $TREF_J(P)$ can perform any trace of P with the additional possibility of performing the event NEW after which it deadlocks.

$$TREF_J(P) \equiv (P[[R_J]]) \,[\![\Sigma - \{NEW\}]\!]\, (CHAOS_{(\Sigma - \{NEW\})} \,[\![\Sigma]\!]\, NEWSTOP)$$

$TREF_J(P)$ can perform any trace of P, but it can also perform NEW when P can do an event from J, after which it deadlocks. It refuses NEW when P can refuse the whole of J.

$$Q \text{ RESPONDSTOLIVE } P \;\; on \;\; J$$

if and only if

$$TREF_J(P) \sqsubseteq_{\mathcal{F}} TREF_J(P \,[\![J]\!]\, LQ)$$

The above refinement will be true, if $P [\![J]\!] Q]$ can refuse all of J then P can refuse all of J. $TREF_J(P [\![J]\!] LQ)$ refuse NEW when $P [\![J]\!] Q]$ refuses J.

Consider an example to illustrate responsiveness of

$$Q = a \rightarrow Q$$

$$P = a \rightarrow b \rightarrow P$$

Q RESPONDSTOLIVE P on $\{a\}$, but Q RESPONDSTOLIVE P on $\{a, b\}$ is not true.

A.4 Type correctness

$\boxed{\text{case: STOP}}$
$(traces_M(\text{STOP}), deadlocks_M(\text{STOP}), revivals_M(\text{STOP})) = (\{\langle\rangle\}, \{\langle\rangle\}, \{\})$ trivially satisfies the conditions T1, D1, R1-R3, RRS05. Hence $(traces_M(\text{STOP}), deadlocks_M(\text{STOP}), revivals_M(\text{STOP}))$ is healthy.

$\boxed{\text{case: SKIP}}$
$(traces_M(\text{SKIP}), deadlocks_M(\text{SKIP}), revivals_M(\text{SKIP})) = (\{\langle\rangle, \langle\checkmark\rangle\}, \{\}, \{\})$ trivially satisfies the conditions T1, D1, R1-R3, RRS05.

$\boxed{\text{case: } ?x : A \rightarrow P}$
Assuming $(traces_M(P(x)), deadlocks_M(P(x)), revivals_M(P(x)))$ is healthy for all $x \in A$, we prove that
$(traces_M(?x : A \rightarrow P(x)), deadlocks_M(?x : A \rightarrow P(x)), revivals_M(?x : A \rightarrow P(x)))$ is also healthy,
where

$$\begin{array}{lcl}
\mathit{traces}(?\ x : A \rightarrow P(x)) & = & \{\langle\rangle\} \cup \{\langle x\rangle \frown t' \mid t' \in \mathit{traces}_M(P(x)), x \in A\} \\
\mathit{deadlocks}(?\ x : A \rightarrow P(x)) & = & \{\langle x\rangle \frown t' \mid t' \in \mathit{deadlocks}_M(P(x)), x \in A\} \\
\mathit{revivals}_M(?\ x : A \rightarrow P(x)) & = & \{(\langle\rangle, X, a) \mid A \cap X = \emptyset \wedge a \in A\} \\
 & & \cup \{(\langle x\rangle \frown t', X, b) \mid (t', X, b) \in \mathit{revivals}_M(P(x)), x \in A\}
\end{array}$$

T1

By definition of $\mathit{traces}_M(?x : A \rightarrow P(x))$, we have $\langle\rangle \in \mathit{traces}_M(?x : A \rightarrow P(x))$. Hence $\mathit{traces}_M(?x : A \rightarrow P(x))$ is non-empty.

We prove that $\mathit{traces}_M(?x : A \rightarrow P(x))$ is prefix closed.
Suppose $\langle a\rangle \frown t' \in \mathit{traces}_M(?x : A \rightarrow P(x))$ such that $t' \in \mathit{traces}_M(P(a))$ and $a \in A$.
Suppose s' is a prefix of $\langle a\rangle \frown t'$.

We consider below two cases.

case 1 : If $s' = \langle\rangle$, then by definition of $\mathit{traces}_M(?x : A \rightarrow P(x))$, $\langle\rangle \in \mathit{traces}_M(?x : A \rightarrow P(x))$.

case 2 : If $s' \neq \langle\rangle$, then s' has the form $\langle a\rangle \frown s''$ and s'' is a prefix of t'. As $\mathit{traces}_M(P(a))$ is prefix closed, we have $s'' \in \mathit{traces}_M(P(a))$. Therefore by definition of $\mathit{traces}_M(?x : A \rightarrow P(x))$, $\langle a\rangle \frown s'' \in \mathit{traces}_M(?x : A \rightarrow P(x))$.

Hence it satisfies $T1$.

D1

Suppose $\langle a\rangle \frown t' \in \mathit{deadlocks}_M(?x : A \rightarrow P(x))$, then $t' \in \mathit{deadlocks}_M(P(a))$ and $a \in A$.
By assumption, we know that $\forall\ s . s \in \mathit{deadlocks}_M(P(a)) \rightarrow s \in \mathit{traces}_M(P(a))$. Thus $t' \in \mathit{traces}_M(P(a))$.
By definition of $\mathit{traces}_M(?x : A \rightarrow P(x))$, $\langle a\rangle \frown t' \in \mathit{traces}_M(?x : A \rightarrow P(x))$.
Hence it satisfies $D1$.

R1

We consider the two clauses separately.

case 1 : Suppose $(\langle\rangle, X, a) \in \mathit{revivals}_M(?x : A \rightarrow P(x))$, then $a \in A$ and $X \cap A \neq \emptyset$. By definition of $\mathit{traces}_M(?x : A \rightarrow P(x))$, we know that $\langle a\rangle \in \mathit{traces}_M(?x : A \rightarrow P(x))$ as $\langle\rangle \in \mathit{traces}_M(P(a))$ and $a \in A$.

case 2 : Suppose $(\langle a\rangle \frown t', X, b) \in \mathit{revivals}_M(?x : A \rightarrow P(x))$, then $(t', X, b) \in \mathit{revivals}_M(P(a))$ and $a \in A$. By assumption we know that $t' \frown \langle b\rangle \in \mathit{traces}_M(P(a))$. By definition of $\mathit{traces}_M((?x : A \rightarrow P(x))$, we know that $\langle a\rangle \frown t' \frown \langle b\rangle \in \mathit{traces}_M(?x : A \rightarrow P(x))$.

Hence it satisfies $R1$.

R2

We consider the two clauses separately.

case 1 : Suppose $(\langle\rangle, X, a) \in \mathit{revivals}_M(?x : A \rightarrow P(x))$, then $a \in A$ and $X \cap A \neq \emptyset$. Suppose $Y \subseteq X$. $Y \subseteq X$ implies that $(\langle\rangle, Y, a) \in \mathit{revivals}_M(?x : A \rightarrow P(x))$, as $a \notin A$ and $Y \cap A \neq \emptyset$.

case 2 : Suppose $(\langle a\rangle \frown t', X, b) \in \mathit{revivals}_M(?x : A \rightarrow P(x))$, then $(t', X, b) \in \mathit{revivals}_M(P(a))$ and $a \in A$. Suppose $Y \subseteq X$. By assumption $(\langle a\rangle \frown t', Y, b) \in \mathit{revivals}_M(P(a))$. By definition of $\mathit{revivals}_M(?x : A \rightarrow P(x))$, it follows that $(\langle a\rangle \frown t', Y, b) \in \mathit{revivals}_M(?x : A \rightarrow P(x))$.

Hence it satisfies $R2$.

R3
We prove that $\forall\ s,\ X,\ a,\ b\ .(s, X, a) \in revivals_M(?x : A \rightarrow P(x)) \wedge b \in \Sigma \rightarrow$
$((s, X, a) \in revivals_M(?x : A \rightarrow P(x)) \vee (s, X \cup \{b\}, a) \in revivals_M(?x : A \rightarrow P(x)))$.
We consider two clauses separately.

case 1: Suppose $(\langle\rangle, X, a) \in revivals_M(?x : A \rightarrow P(x))$, then $a \notin X$ and $X \cap A \neq \emptyset$.

If $b \in A$, then $(\langle\rangle, X, b) \in revivals_M(?x : A \rightarrow P(x))$.

If $b \notin A$, then $(\langle\rangle, X \cup \{b\}, a) \in revivals_M(?x : A \rightarrow P(x))$.

case 2: Suppose $(\langle a\rangle ^\frown t', X, b) \in revivals_M(?x : A \rightarrow P(x))$, then $(t', X, b) \in revivals_M(P(a))$ and $a \in A$. Suppose $c \in \Sigma$. By assumption, we know that $(t', X \cup \{c\}, a) \in revivals_M(P(a))$ or $(t', X, c) \in revivals_M(P(a))$. By definition of $revivals_M(?x : A \rightarrow P(x))$, this leads to $(\langle a\rangle ^\frown t', X \cup \{c\}, a) \in revivals_M(?x : A \rightarrow P(x))$ or
$(\langle a\rangle ^\frown t', X, c) \in revivals_M(?x : A \rightarrow P(x))$.

Hence it satisfies $R3$.

RRS05
We prove that $\forall\ s,\ X\ , a\ .(s, X, a) \in revivals_M(?x : A \rightarrow P(x)) \rightarrow a \notin X$.

We consider two clauses separately.

case 1 : Suppose $(\langle\rangle, X, a) \in revivals_M(?x : A \rightarrow P(x))$. Then $a \in A$ and $X \cap A \neq \emptyset$. This implies that $a \notin X$.

case 2 : Suppose $(\langle a\rangle ^\frown t', X, b) \in revivals_M(?x : A \rightarrow P(x))$, then $(t', X, b) \in revivals_M(P(a))$ and $a \in A$. By assumption, it follows that $b \notin X$.

Hence it satisfies $RRS05$.

$\boxed{\text{case: } P \,\square\, Q}$

Assuming that $(traces_M(P), deadlocks_M(P), revivals_M(P))$ and $(traces_M(Q), deadlocks_M(Q), revivals_M($
are healthy, we prove that $(traces_M(P \,\square\, Q), deadlocks_M(P \,\square\, Q), revivals_M(P \,\square\, Q))$ is healthy where

$$\begin{array}{lcl}
traces(P \,\square\, Q) & = & traces_M(P) \cup traces_M(Q) \\
deadlocks(P \,\square\, Q) & = & ((deadlocks_M(P) \cup deadlocks_M(Q)) \cap \{s \mid s \neq \langle\rangle\}) \\
 & & \cup(deadlocks_M(P) \cap deadlocks_M(Q)) \\
revivals_M(P \,\square\, Q) & = & \{(\langle\rangle, X, a) \mid (\langle\rangle, X) \in failures^b_M(P) \cap failures^b_M(Q) \\
 & & \wedge\ (\langle\rangle, X, a) \in revivals_M(P) \cup revivals_M(Q)\} \\
 & & \cup\{(s, X, a) \mid (s, X, a) \in revivals_M(P) \cup revivals_M(Q) \wedge s \neq \langle\rangle\}
\end{array}$$

where

$$\begin{array}{lcl}
failures^b_M(P) & = & \{(s, X) \mid X \subseteq \Sigma \wedge s \in deadlocks_M(P)\} \\
 & & \cup\{(s, X) \mid (s, X, a) \in revivals_M(P)\}
\end{array}$$

T1, D1
This follows trivially from assumption.

R1
We consider two clauses separately

case 1 : Suppose $(\langle\rangle, X, a) \in \mathit{revivals}_M(P \Box Q)$. By definition, we know that $(\langle\rangle, X, a) \in \mathit{revivals}_M(P)$ or $(\langle\rangle, X, a) \in \mathit{revivals}_M(Q)$. By assumption, we know that $\langle a\rangle \in \mathit{traces}_M(P)$ or $\langle a\rangle \in \mathit{traces}_M(Q)$. By definition of $\mathit{traces}(P \Box Q)$, $\langle a\rangle \in \mathit{traces}_M(P \Box Q)$.

case 2 : Suppose $(s, X, a) \in \mathit{revivals}_M(P \Box Q)$ and $s \neq \langle\rangle$. By definition, we know that $(s, X, a) \in \mathit{revivals}_M(P)$ or $(s, X, a) \in \mathit{revivals}_M(Q)$. By assumption, we know that $s \frown \langle a\rangle \in \mathit{traces}_M(P)$ or $s \frown \langle a\rangle \in \mathit{traces}_M(Q)$. By definition of $\mathit{traces}(P \Box Q)$, $s \frown \langle a\rangle \in \mathit{traces}_M(P \Box Q)$.

Hence it satisfies R1.

R2
We consider two clauses separately.

case 1 : Suppose $(\langle\rangle, X, a) \in \mathit{revivals}_M(P \Box Q)$. Suppose $Y \subseteq X$. By definition, we know that $(\langle\rangle, X, a) \in \mathit{revivals}_M(P)$ or $(\langle\rangle, X, a) \in \mathit{revivals}_M(Q)$. By assumption, we know that $(\langle\rangle, Y, a) \in \mathit{revivals}_M(P)$ or $(\langle\rangle, Y, a) \in \mathit{revivals}_M(Q)$. By definition of $\mathit{revivals}_M(P \Box Q)$, we have $(\langle\rangle, X) \in \mathit{failures}^b_M(P)$ and $(\langle\rangle, X) \in \mathit{failures}^b_M(Q)$. Since P and Q are healthy, by Lemma 3.2, $(\langle\rangle, Y) \in \mathit{failures}^b_M(P)$ and $(\langle\rangle, Y) \in \mathit{failures}^b_M(Q)$. Therefore, $(\langle\rangle, Y, a) \in \mathit{revivals}_M(P \Box Q)$.

case 2 : Suppose $(s, X, a) \in \mathit{revivals}_M(P \Box Q)$ and $s \neq \langle\rangle$. Suppose $Y \subseteq X$. By definition, we know that $(s, X, a) \in \mathit{revivals}_M(P)$ or $(s, X, a) \in \mathit{revivals}_M(Q)$. By assumption, we know that $(\langle\rangle, Y, a) \in \mathit{revivals}_M(P)$ or $\langle\rangle, Y, a) \in \mathit{revivals}_M(Q)$.
Therefore, $(\langle\rangle, Y, a) \in \mathit{revivals}_M(P \Box Q)$.

Hence it satisfies R2.

R3
We prove that $\forall\ s,\ X\ , a,\ b\ .(s, X, a) \in \mathit{revivals}_M(P \Box Q) \wedge b \in \Sigma \rightarrow$
$(s, X, b) \in \mathit{revivals}_M(P \Box Q) \vee (s, X \cup \{b\}, a) \in \mathit{revivals}_M(P \Box Q)$.

We consider only the first clause as the proof for the second clause follows trivially from the assumption.
Suppose $(\langle\rangle, X, a) \in \mathit{revivals}_M(P \Box Q)$. By definition, we know that $(\langle\rangle, X, a) \in \mathit{revivals}_M(P)$ or $(\langle\rangle, X, a) \in \mathit{revivals}_M(Q)$.

With out loss of generality, consider $(\langle\rangle, X, a) \in \mathit{revivals}_M(P)$ case only. By definition of $\mathit{revivals}_M(P \Box Q)$, we also know $(\langle\rangle, X) \in \mathit{failures}^b_M(P)$ and $(\langle\rangle, X) \in \mathit{failures}^b_M(Q)$.

Suppose b to be an arbitrary event in Σ.

By assumption, we know that $(\langle\rangle, X, b) \in \mathit{revivals}_M(P)$ or $(\langle\rangle, X \cup \{b\}, a) \in \mathit{revivals}_M(P)$. Hence we consider two cases.

case 1: $(\langle\rangle, X, b) \in \mathit{revivals}_M(P)$.
If $(\langle\rangle, X, b) \in \mathit{revivals}_M(P)$, then it is clear that $(\langle\rangle, X, b) \in \mathit{revivals}_M(P \Box Q)$ as $(\langle\rangle, X) \in \mathit{failures}^b_M(P)$ and $(\langle\rangle, X) \in \mathit{failures}^b_M(Q)$ are satisfied by assumption.

case 2: $(\langle\rangle, X \cup \{b\}, a) \in \mathit{revivals}_M(P)$.
We have to prove $(\langle\rangle, X \cup \{b\}, a) \in \mathit{revivals}_M(P \Box Q)$ or $(\langle\rangle, X, b) \in \mathit{revivals}_M(P \Box Q)$.

To this end we have to prove $(\langle\rangle, X \cup \{b\}) \in \mathit{failures}^b_M(P)$ and $(\langle\rangle, X \cup \{b\}) \in \mathit{failures}^b_M(Q)$ are satisfied or that $(\langle\rangle, X) \in \mathit{failures}^b_M(P)$ and $(\langle\rangle, X) \in \mathit{failures}^b_M(Q)$ are satisfied.

By expanding the definition of $\mathit{failures}^b_M(Q)$, we consider the two cases below.

case 2.1: $\langle\rangle \in \mathit{deadlocks}_M(Q)$. By definition of $\mathit{failures}^b_M$, $(\langle\rangle, X \cup \{b\}) \in \mathit{failures}^b_M(Q)$. By definition of $\mathit{failures}^b_M(P)$, $(\langle\rangle, X \cup \{b\}, a) \in \mathit{revivals}_M(P)$ implies
$(\langle\rangle, X \cup \{b\}) \in \mathit{failures}^b_M(P)$.
Hence $(\langle\rangle, X \cup \{b\}, a) \in \mathit{revivals}_M(P \,\Box\, Q)$.

case 2.2: $(\langle\rangle, X, a') \in \mathit{revivals}_M(Q)$. Since Q is healthy, $(\langle\rangle, X, b) \in \mathit{revivals}_M(Q)$ or $(\langle\rangle, X \cup \{b\}, a') \in \mathit{revivals}_M(Q)$. We consider below two cases.

case 2.2.1: If $(\langle\rangle, X \cup \{b\}, a') \in \mathit{revivals}_M(Q)$, then $(\langle\rangle, X \cup \{b\}) \in \mathit{failures}^b_M(Q)$. $(\langle\rangle, X \cup \{b\}, a) \in \mathit{revivals}_M(P)$ implies $(\langle\rangle, X \cup \{b\}) \in \mathit{failures}^b_M(P)$. Hence $(\langle\rangle, X \cup \{b\}, a) \in \mathit{revivals}_M(P \,\Box\, Q)$.

case 2.2.2: $(\langle\rangle, X, b) \in \mathit{revivals}_M(Q)$
$(\langle\rangle, X, b) \in \mathit{revivals}_M(Q)$ implies $(\langle\rangle, X) \in \mathit{failures}^b_M(Q)$. $(\langle\rangle, X) \in \mathit{failures}^b_M(P)$ follows from the initial assumption $(\langle\rangle, X, a) \in \mathit{revivals}_M(P)$.
Hence $(\langle\rangle, X, b) \in \mathit{revivals}_M(P \,\Box\, Q)$.

Hence it satisfies R3.

RRS05
We have to show that $\forall\ s,\ X\ , a\ .\ (s, X, a) \in \mathit{revivals}_M(P \,\Box\, Q) \rightarrow a \notin X$. This directly follows from the assumption of $RRS05$ on processes P and Q.

case: $P \,|[\, X \,]|\, Q$

Assuming that $(\mathit{traces}(P), \mathit{deadlocks}(P), \mathit{revivals}(P))$ and $(\mathit{traces}(Q), \mathit{deadlocks}(Q), \mathit{revivals}(Q))$ are healthy, we prove that $(\mathit{traces}(P \,|[\, X \,]|\, Q), \mathit{deadlocks}(P \,|[\, X \,]|\, Q), \mathit{revivals}(P \,|[\, X \,]|\, Q))$ is also healthy, where

$$\begin{array}{lcl}
\mathit{traces}(P \,|[\, X \,]|\, Q) & = & \{t_1 \,|[\, X \,]|\, t_2 \mid t_1 \in \mathit{traces}(P),\ t_2 \in \mathit{traces}(Q)\} \\
\mathit{deadlocks}(P \,|[\, X \,]|\, Q) & = & \{u \mid (s, Y) \in \mathit{failures}(P), (t, Z) \in \mathit{failures}(Q)\ . \\
 & & Y - (X \cup \{\checkmark\}) = Z - (X \cup \{\checkmark\}) \\
 & & \wedge\ u \in s \,|[\, X \,]|\, t \\
 & & \wedge\ \Sigma^{\checkmark} = Y \cup Z\} \\
\mathit{revivals}(P \,|[\, X \,]|\, Q) & = & \{(u, Y \cup Z), a) \mid \\
 & & \exists\, s, t.(s, Y) \in \mathit{failures}(P) \wedge (t, Z) \in \mathit{failures}(Q) \wedge u \in u \in s \,|[\, X \,]|\, t \cap \Sigma^* \\
 & & \wedge\ Y - (X \cup \{\checkmark\}) = Z - (X \cup \{\checkmark\}) \\
 & & \wedge\ ((a \in X \wedge (s, Y, a) \in \mathit{revivals}(P) \wedge (t, Z, a) \in \mathit{revivals}(Q)) \\
 & & \ \vee\ a \notin X \wedge (s, Y, a) \in \mathit{revivals}(P) \\
 & & \ \vee\ a \notin X \wedge (t, Z, a) \in \mathit{revivals}(Q)))) \ \}.
\end{array}$$

T1
$\mathit{traces}(P \,|[\, X \,]|\, Q)$ is non-empty as $\langle\rangle \,|[\, X \,]|\, \langle\rangle = \{\langle\rangle\}$ as $\langle\rangle \in \mathit{traces}(P)$ and $\langle\rangle \in \mathit{traces}(Q)$.

We prove that $\{t_1 \,|[X]|\, t_2 \mid t_1 \in \mathit{traces}(P),\ t_2 \in \mathit{traces}(Q)\})$ is prefix closed. We prove by induction on t in $s \,|[\, X \,]|\, t$ where $s \in \mathit{traces}(P)$ and $t \in \mathit{traces}(Q)$. We consider the following cases.

case 1: $\langle\rangle$
: $\langle\rangle \,|[\, X \,]|\, \langle\rangle = \{\langle\rangle\}$ is prefix closed.

case 2: $\langle x\rangle \in X$
: $\langle\rangle \,|[\, X \,]|\, \langle x\rangle = \{\}$ is prefix closed.

case 3: $\langle y\rangle \notin X$
: $\langle\rangle \,|[\, X \,]|\, \langle y\rangle = \{\langle y\rangle\}$ is prefix closed as $traces(Q)$ and $traces(P)$ are prefix closed we have $\langle\rangle \in traces(Q)$ and $\langle\rangle \in traces(P)$ by the case (i), we have also $\langle\rangle \in traces(P)|[X]|traces(Q)$. Hence it is prefix closed.

case 4: $\langle x\rangle \frown s \,|[\, X \,]|\, \langle y\rangle \frown t = \{\langle y\rangle \frown u \mid u \in \langle x\rangle \frown s \,|[\, X \,]|\, t\}$
: We know by induction hypothesis that $\langle x\rangle \frown s \,|[\, X \,]|\, t$ is prefix closed, hence $\{\langle y\rangle \frown u \mid u \in \langle x\rangle \frown s \,|[\, X \,]|\, t\}$ is also prefix closed. The proofs for the other cases are similar.

D1:
First we prove that $(s, X) \in failures(P)$ implies $s \in traces(P)$ by healthiness condition of $T1$, $D1$, and $R1$. A trace s comes either from a revival or a deadlock.

If $(s, X, a) \in revivals(P)$ for some X and a, then we know by $R1$ that $s \frown \langle a\rangle \in traces(P)$. By $T1$, it follows that $s \in traces(P)$.

If $s \in deadlocks(P)$, then by $D1$, it follows that $s \in traces(P)$.

It is clear that $deadlocks(P \,|[\, X \,]|\, Q) \subseteq traces(P \,|[\, X \,]|\, Q)$ as all the traces of $deadlocks(P \,|[\, X \,]|\, Q)$ is also the $traces(P \,|[X]|\, Q)$ as $deadlocks(P \,|[X]|\, Q)$ has more conditions on $s \,|[X]|\, t$ and $s \in traces(P)$ and $t \in traces(Q)$ comes from $failures(P)$ and $failures(Q)$ respectively.

R1
We assume that P and Q satisfies $R1$. We consider the following two cases

case 1: $a \in X$

Suppose $(u, Y \cup Z, a) \in revivals(P \,|[\, X \,]|\, Q)$. We know that

$$\begin{aligned}&\exists\, s, t.(s, Y) \in failures(P) \wedge (t, Z) \in failures(Q) \wedge\\ &u \in s \,|[\, X \,]|\, t \wedge\\ &Y - (X \cup \{\checkmark\}) = Z - (X \cup \{\checkmark\}) \wedge\\ &(a \in X \wedge (s, Y, a) \in revivals(P) \wedge\\ &\quad (t, Z, a) \in revivals(P))\end{aligned}$$

We know by assumption that $(s, Y, a) \in revivals(P) \rightarrow s \frown \langle a\rangle \in traces(P)$ and $(t, Z, a) \in revivals(Q) \rightarrow t \frown \langle a\rangle \in traces(Q)$. By the definition of $s \,|[\, X \,]|\, t$, $s \frown \langle a\rangle \,|[\, X \,]|\, t \frown \langle a\rangle = \{u \frown \langle a\rangle \mid u \in s \,|[\, X \,]|\, t\}$ as we know that for $a \in X$ and $\langle a\rangle \,|[\, X \,]|\, \langle a\rangle = \{\langle a\rangle\}$. Hence we know that $(u, Y \cup Z, a) \in revivals(P \,|[\, X \,]|\, Q) \rightarrow u \frown \langle a\rangle \in traces(P \,|[\, X \,]|\, Q)$.

case 2: $a \notin X$

Suppose $(u, Y \cup Z, a) \in revivals(P \,|[\, X \,]|\, Q)$ and $a \notin X$. We know that there exists

$$\begin{aligned}&\exists\, s, t.(s, Y) \in failures(P) \wedge (t, Z) \in failures(Q) \wedge\\ &u \in s \,|[\, X \,]|\, t \wedge\\ &Y - (X \cup \{\checkmark\}) = Z - (X \cup \{\checkmark\}) \wedge\\ &(a \notin X \wedge ((s, Y, a) \in revivals(P) \vee\\ &\quad (t, Z, a) \in revivals(Q)))\}).\end{aligned}$$

We consider two cases

case 2.1 : $(s, Y, a) \in \mathit{revivals}(P)$.
As P satisfies $R1$ by assumption, we obtain $s \frown \langle a \rangle \in \mathit{traces}(P)$. By the definition of $s \,|[X]|\, t$, $s \frown \langle a \rangle \,|[X]|\, t = \{u \frown \langle a \rangle \mid u \in s \,|[X]|\, t\}$ as $\langle\rangle \,|[X]|\, \langle a \rangle = \{\langle a \rangle\}$. Hence $(u, Y \cup Z, a) \in \mathit{revival}(P \,|[X]|\, Q) \to u \frown \langle a \rangle \in \mathit{traces}(P \parallel Q))$

case 2.2: $(t, Z, a) \in \mathit{revivals}(Q)$.
As Q satisfies $R2$ by assumption, we obtain $t \frown \langle a \rangle \in \mathit{traces}(Q)$. By the definition of $s \,|[X]|\, t$, $s \,|[X]|\, t \frown \langle a \rangle = \{u \frown \langle a \rangle \mid u \in s \,|[X]|\, t\}$ as $\langle\rangle \,|[X]|\, \langle a \rangle = \{\langle a \rangle\}$. Hence $(u, Y \cup Z, a) \in \mathit{revivals}(P \,|[X]|\, Q) \to u \frown \langle a \rangle \in \mathit{traces}(P \,|[X]|\, Q)$.

$(\mathit{traces}(P \,|[X]|\, Q), \mathit{deadlocks}(P \,|[X]|\, Q), \mathit{revivals}(P \,|[X]|\, Q))$ satisfies $R1$.

R2
We assuming that P and Q satisfies $R2$. We prove that
$(s, Y \cup Z, a) \in \mathit{revivals}(P \,|[X]|\, Q) \wedge M \subseteq Y \cup Z \to (s, M, a) \in \mathit{revival}(P \,|[X]|\, Q$.

We know by Lemma 6.2 that $(s, X) \in \mathit{failures}(P) \wedge Y \subseteq X \to (s, Y) \in \mathit{failures}(Q)$.

Suppose $(u, Y \cup Z, a) \in \mathit{revivals}(P \,|[X]|\, Q)$. We know by definition of $\mathit{revivals}(P \,|[X]|\, Q)$,

$$\begin{array}{l}
\exists\, s, t.(s, Y) \in \mathit{failures}(P) \wedge (t, Z) \in \mathit{failures}(Q) \wedge \\
u \in s \,|[X]|\, t \wedge \\
Y - (X \cup \{\checkmark\}) = Z - (X \cup \{\checkmark\}) \wedge \\
(a \in X \wedge (s, Y, a) \in \mathit{revivals}(P) \wedge \\
\quad (t, Z, a) \in \mathit{revivals}(P)) \vee \\
(a \notin X \wedge ((s, Y, a) \in \mathit{revivals}(P) \vee \\
\quad (t, Z, a) \in \mathit{revivals}(Q)))\})
\end{array}$$

By lemma 6.2, we know that there exists $(s, Y \cap M) \in \mathit{failures}(P) \wedge (t, Z \cap M) \in \mathit{failures}(Q)$ as $Y \cap M \subseteq Y$ and $Z \cap M \subseteq Z$.

It is clear that it satisfies all the following condition by Lemma 6.2 and by assumption

$$\begin{array}{l}
(s, Y \cap M) \in \mathit{failures}(P) \wedge (t, Z \cap M) \in \mathit{failures}(Q) \wedge \\
u \in s \,|[X]|\, t \wedge \\
(Y \cap M) - (X \cup \{\checkmark\}) = (Z \cap M)) - (X \cup \{\checkmark\}) \wedge \\
(a \in X \wedge (s, Y \cap M, a) \in \mathit{revivals}(P) \wedge \\
\quad (t, Z \cap M, a) \in \mathit{revivals}(P)) \vee \\
(a \notin X \wedge ((s, Y \cap M, a) \in \mathit{revivals}(P) \vee \\
\quad (t, Z \cap M, a) \in \mathit{revivals}(Q)))\})
\end{array}$$

as $\mathit{revivals}(P)$ and $\mathit{revivals}(P)$ satisfies R2.
$(\mathit{traces}(P \,|[X]|\, Q), \mathit{deadlocks}(P \,|[X]|\, Q), \mathit{revivals}(P \,|[X]|\, Q))$ satisfies $R2$.

R3
We assume that P and Q satisfies $R3$. We prove that
$(u, Y \cup Z, a) \in \mathit{revivals}(P \,|[X]|\, Q) \wedge c \in \Sigma \to$
$\quad (u, Y \cup Z, c) \in \mathit{revivals}(P \,|[X]|\, Q) \vee (s, Y \cup Z \cup \{c\}, a) \in \mathit{revivals}(P \,|[X]|\, Q)$.

Suppose $(u, Y \cup Z, a) \in \mathit{revivals}(P \,|[\, X \,]|\, Q)$. Then we know that
$(s, Y) \in \mathit{failures}(P) \wedge (t, Z) \in \mathit{failures}(Q) \wedge u \in u \in s \,|[\, X \,]|\, t \cap \Sigma^*$
$\wedge\ Y - (X \cup \{\checkmark\}) = Z - (X \cup \{\checkmark\})$
$\wedge\ ((a \in X \wedge (s, Y, a) \in \mathit{revivals}(P) \wedge (t, Z, a) \in \mathit{revivals}(Q))$
$\vee\ a \notin X \wedge (s, Y, a) \in \mathit{revivals}(P)$
$\vee\ a \notin X \wedge (t, Z, a) \in \mathit{revivals}(Q))))$.

We consider the following three cases:

case 1: $((a \in X \wedge (s, Y, a) \in \mathit{revivals}(P) \wedge (t, Z, a) \in \mathit{revivals}(Q))$

We know from hypothesis of $R3$ that
$(s, Y, c) \in \mathit{revivals}(P) \vee (s, Y \cup \{c\}, a) \in \mathit{revivals}(P)$.
$(t, Z, c) \in \mathit{revivals}(Q) \vee (t, Z \cup \{c\}, a) \in \mathit{revivals}(Q)$.
We consider the following four cases

case 1.1: If $(s, Y, c) \in \mathit{revivals}(P)$ and $(s, Z, c) \in \mathit{revivals}(Q)$.

case 1.1.1: If $c \in C$, then it satisfies the following condition:
$\exists\, s, t.(s, Y) \in \mathit{failures}(P) \wedge (t, Z) \in \mathit{failures}(Q) \wedge$
$u \in s \,|[\, X \,]|\, t \wedge$
$Y - (X \cup \{\checkmark\}) = Z - (X \cup \{\checkmark\}) \wedge$
$(c \in X \wedge (s, Y, c) \in \mathit{revivals}(P) \wedge$
$(t, Z, c) \in \mathit{revivals}(P))$
Hence it satisfies $(u, Z \cup Y, c) \in \mathit{revivals}(P \,|[\, X \,]|\, Q)$.

case 1.1.2: If $c \notin C$, then it satisfies the following condition:
$(s, Y) \in \mathit{failures}(P) \wedge (t, Z) \in \mathit{failures}(Q) \wedge$
$u \in s \,|[\, X \,]|\, t \wedge$
$Y - (X \cup \{\checkmark\}) = Z - (X \cup \{\checkmark\}) \wedge$
$(c \notin X \wedge (s, Y, c) \in \mathit{revivals}(P) \vee$
$(t, Z, c) \in \mathit{revivals}(P))$
Hence it satisfies $(u, Z \cup Y, c) \in \mathit{revivals}(P \,|[\, X \,]|\, Q)$.

case 1.2: If $a \in X$, $(s, Y \cup \{c\}, a) \in \mathit{revivals}(P)$ and $(t, Z, c) \in \mathit{revivals}(Q)$, then we know that

case 1.2.1: $c \notin X$, then following condition satisfies
$\exists\, s, t.(s, Y) \in \mathit{failures}(P) \wedge (t, Z) \in \mathit{failures}(Q) \wedge$
$u \in s \,|[\, X \,]|\, t \wedge$
$Y - (X \cup \{\checkmark\}) = Z - (X \cup \{\checkmark\}) \wedge$
$(c \notin X \wedge (s, Y, c) \in \mathit{revivals}(P) \vee$
$(t, Z, c) \in \mathit{revivals}(P))$.
Hence it satisfies $(u, Z \cup Y, c) \in \mathit{revivals}(P \,|[\, X \,]|\, Q)$.

case 1.2.2: $c \in X$, then following condition satisfies
$(s, Y \cup \{c\}) \in \mathit{failures}(P) \wedge (t, Z) \in \mathit{failures}(Q) \wedge$
$u \in s \,|[\, X \,]|\, t \wedge$
$Y \cup \{c\} - (X \cup \{\checkmark\}) = Z - (X \cup \{\checkmark\}) \wedge$
$((a \in X \wedge (s, Y \cup \{c\}, a) \in \mathit{revivals}(P) \wedge (t, Z, a) \in \mathit{revivals}(Q)))$
Hence it satisfies $(u, Z \cup Y \cup \{c\}, a) \in \mathit{revivals}(P \,|[\, X \,]|\, Q)$.

case 1.3: If $a \in X$, $(s, Y, a) \in \mathit{revivals}(P)$ and $(t, Z \cup \{c\}, c) \in \mathit{revivals}(Q)$, then we know that

case 1.3.1: $c \notin X$, then following condition satisfies

$$\begin{array}{l} (s, Y) \in \mathit{failures}(P) \wedge (t, Z) \in \mathit{failures}(Q) \wedge \\ u \in s \,\|[X]\|\, t \wedge \\ Y - (X \cup \{\checkmark\}) = Z - (X \cup \{\checkmark\}) \wedge \\ (c \notin X \wedge (s, Y, c) \in \mathit{revivals}(P) \vee \\ \quad (t, Z, c) \in \mathit{revivals}(P)) \end{array}$$

Hence it satisfies $(u, Z \cup Y, c) \in \mathit{revivals}(P \,\|[X]\|\, Q)$.

case 1.3.2: $c \in X$, then following condition satisfies

$$\begin{array}{l} (s, Y \cup \{c\}) \in \mathit{failures}(P) \wedge (t, Z) \in \mathit{failures}(Q) \wedge \\ u \in s \,\|[X]\|\, t \wedge \\ Y \cup \{c\} - (X \cup \{\checkmark\}) = Z - (X \cup \{\checkmark\}) \wedge \\ \quad ((a \in X \wedge (s, Y \cup \{c\}, a) \in \mathit{revivals}(P) \wedge (t, Z, a) \in \mathit{revivals}(Q))) \end{array}$$

Hence it satisfies $(u, Z \cup Y \cup \{c\}, a) \in \mathit{revivals}(P \,\|[X]\|\, Q)$.

case 1.4: If $a \in X$, $(s, Y \cup \{c\}, a) \in \mathit{revivals}(P)$ and $(t, Z \cup \{c\}, a) \in \mathit{revivals}(Q)$, then we know that it satisfies the below condition

$$\begin{array}{l} (s, Y \cup \{c\}) \in \mathit{failures}(P) \wedge (t, Z \cup \{c\}) \in \mathit{failures}(Q) \wedge \\ u \in s \,\|[X]\|\, t \wedge \\ Y \cup \{c\} - (X \cup \{\checkmark\}) = Z \cup \{c\} - (X \cup \{\checkmark\}) \wedge \\ ((s, Y \cup \{c\}, a) \in \mathit{revivals}(P) \wedge (t, Z \cup \{c\}, a) \in \mathit{revivals}(P)) \end{array}$$

Hence it satisfies $(u, Z \cup Y \cup \{c\}, a) \in \mathit{revivals}(P \,\|[X]\|\, Q)$.

case 2: $a \notin X \wedge (s, Y, a) \in \mathit{revivals}(P)$

We know from hypothesis of $R3$ that

$(s, Y, c) \in \mathit{revivals}(P) \vee (s, Y \cup \{c\}, a) \in \mathit{revivals}(P)$.

We consider the following two cases

case 2.1: If $(s, Y, c) \in \mathit{revivals}(P)$, then we consider following two cases

case 2.1.1: If $c \notin X$, then it satisfies the following condition

$$\begin{array}{l} (s, Y) \in \mathit{failures}(P) \wedge (t, Z) \in \mathit{failures}(Q) \wedge \\ u \in s \,\|[X]\|\, t \wedge \\ Y - (X \cup \{\checkmark\}) = Z - (X \cup \{\checkmark\}) \wedge \\ (c \notin X \wedge ((s, Y, c) \in \mathit{revivals}(P) \vee (t, Z, c) \in \mathit{revivals}(P))) \end{array}$$

Hence it satisfies $(u, Z \cup Y, c) \in \mathit{revivals}(P \,\|[X]\|\, Q)$.

case 2.1.2: If $c \in X$, then we expand the definition of $(t, Z) \in \mathit{failures}(Q)$ consider the following cases:

case 2.1.2.1: $t \in \mathit{deadlocks}(Q)$, then we know that $(t, Z \cup \{c\}) \in \mathit{failures}(Q)$. Hence it satisfies

$$\begin{array}{l} (s, Y) \in \mathit{failures}(P) \wedge (t, Z \cup \{c\}) \in \mathit{failures}(Q) \wedge \\ u \in s \,\|[X]\|\, t \wedge Z \cup Y \cup \{c\} = X \cup Y \cup \{c\} \wedge \\ Y - (X \cup \{\checkmark\}) = Z \cup \{c\} - (X \cup \{\checkmark\}) \wedge \\ \quad ((a \notin X \wedge (s, Y, a) \in \mathit{revivals}(P) \vee (t, Z \cup \{a\}, a) \in \mathit{revivals}(Q))). \end{array}$$

Hence it satisfies $(u, Z \cup Y \cup \{c\}, a) \in \mathit{revivals}(P \,\|[X]\|\, Q)$.

case 2.1.2.2: $t ^\frown \langle \checkmark \rangle \in traces(Q)$, then the proof is similar to *case* 2.1.2.1.

case 2.1.2.3: $(t, Z, aa) \in revivals(Q)$, then we know that $(t, Z, c) \in revivals(Q) \lor (t, Z \cup \{c\}, aa) \in revivals(Q)$, then

case 2.1.2.3.1: $(t, Z, c) \in revivals(Q)$, then the below condition satisfies

$$\begin{aligned}
&\exists\, s, t.(s, Y) \in failures(P) \land (t, Z) \in failures(Q) \land \\
&u \in s \,|[X]|\, t \land \\
&Y - (X \cup \{\checkmark\}) = Z - (X \cup \{\checkmark\}) \land \\
&(c \in X \land (s, Y, c) \in revivals(P) \land \\
&\quad (t, Z, c) \in revivals(P))
\end{aligned}$$

. Hence it satisfies $(u, Z \cup Y, c) \in revivals(P \,|[X]|\, Q)$.

case 2.1.2.3.2: If $(t, Z \cup \{c\}, aa) \in revivals(Q)$, then it satisfies the following condition:

$$\begin{aligned}
&(s, Y) \in failures(P) \land (t, Z) \in failures(Q) \land \\
&u \in s \,|[X]|\, t \land \\
&Y - (X \cup \{\checkmark\}) = Z \cup \{c\} - (X \cup \{\checkmark\}) \land \\
&(a \notin X \land (s, Y, a) \in revivals(P) \lor \\
&\quad (t, Z, c) \in revivals(Q)).
\end{aligned}$$

Hence it satisfies $(u, Z \cup Y \cup \{c\}, aa) \in revivals(P \,|[X]|\, Q)$.

case 2.2: If $(s, Y \cup \{c\}, a) \in revivals(P)$, then we consider the following two cases

case 2.2.1: $c \in X$, then the following condition is satisfies

$$\begin{aligned}
&(s, Y \cup \{c\}) \in failures(P) \land (t, Z) \in failures(Q) \land \\
&u \in s \,|[X]|\, t \land \\
&Y \cup \{c\} - (X \cup \{\checkmark\}) = Z - (X \cup \{\checkmark\}) \land \\
&(a \notin X \land (s, Y \cup \{c\}, a) \in revivals(P) \lor \\
&\quad (t, Z, c) \in revivals(Q)).
\end{aligned}$$

Hence it satisfies $(u, Z \cup Y \cup \{c\}, a) \in revivals(P \,|[X]|\, Q)$.

case 2.2.2: $c \notin X$, then we expand the definition of *failures*(Q), consider the following cases

case 2.2.2.1: $t \in deadlocks(Q)$, then we know that $(t, Z \cup \{c\}) \in failures(Q)$. It satisfies the following

$$\begin{aligned}
&(s, Y \cup \{c\}) \in failures(P) \land (t, Z \cup \{c\}) \in failures(Q) \land \\
&u \in s \,|[X]|\, t \land \\
&Y \cup \{c\} - (X \cup \{\checkmark\}) = Z \cup \{c\} - (X \cup \{\checkmark\}) \land \\
&((a \notin X \land (s, Y \cup \{c\}, a) \in revivals(P) \lor (t, Z \cup \{a\}, a) \in revivals(Q))).
\end{aligned}$$

Hence it satisfies $(u, Z \cup Y \cup \{c\}, a) \in revivals(P \,|[X]|\, Q)$.

case 2.2.2.2: $t ^\frown \langle \checkmark \rangle \in traces(Q)$, then the proof is similar to *case* 2.2.2.1.

case 2.2.2.3: $(t, Z, aa) \in revivals(Q)$, then we know that $(t, Z, c) \in revivals(Q) \lor (t, Z \cup \{c\}, aa) \in revivals(Q)$, then

case 2.2.2.3.1: $(t, Z, c) \in revivals(Q)$, then the below condition satisfies

$$\begin{aligned}
&\exists\, s, t.(s, Y) \in failures(P) \land (t, Z) \in failures(Q) \land \\
&u \in s \,|[X]|\, t \land
\end{aligned}$$

$Y - (X \cup \{\checkmark\}) = Z - (X \cup \{\checkmark\}) \wedge$
$(c \notin X \wedge (s, Y, a) \in \mathit{revivals}(P) \vee$
$(t, Z, c) \in \mathit{revivals}(P))$.
Hence it satisfies $(u, Z \cup Y, c) \in \mathit{revivals}(P \,|[\, X \,]|\, Q)$.

case 2.2.2.3.2: If $(t, Z \cup \{c\}, aa) \in \mathit{revivals}(Q)$, then it satisfies the following condition:
$(s, Y \cup \{c\}) \in \mathit{failures}(P) \wedge (t, Z \cup \{c\}) \in \mathit{failures}(Q) \wedge$
$u \in s \,|[\, X \,]|\, t \wedge$
$Y \cup \{c\} - (X \cup \{\checkmark\}) = Z \cup \{c\} - (X \cup \{\checkmark\}) \wedge$
$(a \notin X \wedge (s, Y \cup \{a\}, a) \in \mathit{revivals}(P) \vee$
$(t, Z, c) \in \mathit{revivals}(Q))$.
Hence it satisfies $(u, Z \cup Y \cup \{c\}, a) \in \mathit{revivals}(P \,|[\, X \,]|\, Q)$.

case 3: $a \notin X \wedge (t, Z, a) \in \mathit{revivals}(Q)$
It is similar to *case* 2.

A.5 Syntax and Semantics of CCS

Calculus for Communicating Systems (CCS) was designed to understand concurrency and communication using a few primitive operators. In this section, first we describe the syntax of CCS and then we give the semantics of CCS. The atomic actions in CCS is usually called actions $Act = \mathcal{L} \cup \{\tau\}$. $\mathcal{L} = \mathcal{A} \cup \overline{\mathcal{A}}$ where $\mathcal{A}$ and $\overline{\mathcal{A}}$ are a set of names and co-names respectively. The set of co-name is defined as $\overline{\mathcal{A}} = \{\overline{a} \mid a \in \mathcal{A}\}$. λ ranges over $\mathcal{L}$. $\mathcal{L}$ is similar to the alphabet Σ in CSP. In CCS, a process is called *agents*. We define the set of agents Ψ in CCS. Ψ is the smallest set containing the following agents, where E, E_i are in Ψ:

- $\alpha.E$, a Prefix operator where $\alpha \in Act$. It is similar to prefix operator in CSP.
- $\Sigma_{i \in I} E_i$ a Summation where I is an indexing set. It is a choice operator in CCS. $E_1 + E_2$ is syntacial sugar for $\Sigma_{1,2} E_i$. If I is empty set, $\mathbf{0} = \Sigma_{\{\}}$.
- $E_1 \mid E_2$ is a composition operator in CCS. The action a interact only with $\overline{a}$.
- $E \setminus L$ is a restriction operator where $L \subseteq \mathcal{L}$. The process $E \setminus L$ that cannot perform any actions in L.
- $E[f]$, a Relabelling operator where f is a relabelling function.

A constant is an agent whose meaning is given by a defining equation in the form

$$A \overset{def}{=} P.$$

An example is $A = a.E$. Mutual recursion is defined as follows

$$A \overset{def}{=} a.B \qquad B \overset{def}{=} b.A$$

The names $a \in L$ are bound in $E \setminus L$. The free names of P, denoted $\mathit{fn}(P)$, are names in P that are not bound.

Definition: Structural Congruence, $\equiv$. Structural congruence, $\equiv$ is the least congruence relation on closed terms under the following rules, together with change of bound names and variables (alpha conversion) and reordering of terms in a summation:
1. $P \mid 0 \equiv P, P \mid Q \equiv Q \mid P, P \mid (Q \mid R) \equiv (P \mid Q) \mid R$
2. $P \mid Q \setminus L \equiv P \mid (Q \setminus L)$, if $\alpha \notin fn(P)$ for each $\alpha \in L$.
3. $0 \setminus X \equiv 0$.

The semantics of Ccs is given by a labelled transition systems

$$(S, T, \{\xrightarrow{t}: t \in T\}$$

consists of a set S of states to be agents in Ψ , a set T of transition labels to be action in Act , and a transition relation $\xrightarrow{t} \subseteq S \times S$ for each $t \in T$. The semantics consists of one or two transition rules for each operators and one rule for constants. Each rule is associated with a operator.

$$Act \ \frac{}{\alpha.E \xrightarrow{\alpha} E} \qquad Sum_j \ \frac{E_j \xrightarrow{\alpha} E_j'}{\Sigma_{i \in I} E_i \xrightarrow{\alpha} E_j'} \ (j \in I)$$

$$Com_1 \ \frac{E \xrightarrow{\alpha} E'}{E \mid F \xrightarrow{\alpha} E' \mid F} \qquad Com_2 \ \frac{F \xrightarrow{\alpha} F'}{E \mid F \xrightarrow{\alpha} E \mid F'} \qquad Com_3 \ \frac{E \xrightarrow{l} E' \qquad F \xrightarrow{\bar{l}} F'}{E \mid F \xrightarrow{\tau} E' \mid F'}$$

$$Res \ \frac{E \xrightarrow{\alpha} E'}{E \setminus L \xrightarrow{\alpha} E' \setminus L} \ (\alpha \ \overline{\alpha} \notin L) \qquad Rel \ \frac{E \xrightarrow{\alpha} E'}{E[f] \xrightarrow{f(\alpha)} E'[f]}$$

$$Con \ \frac{P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'} \ A \overset{def}{=} P \qquad Cong \ \frac{P \equiv P' \qquad P \xrightarrow{\alpha} Q' \qquad Q \equiv Q'}{P \xrightarrow{\alpha} Q}$$

Similar to Csp, we can define the internal choice operator as $P \# Q \overset{def}{=} \tau.P + \tau.Q$ We denotes $\tilde{a}$ and $\tilde{\alpha}$ for (possibly empty) sequences of names and actions. For $\tilde{\alpha} = \alpha_0 \ldots \alpha_n$, we denote $P \xrightarrow{\tilde{\alpha}} P'$, if there exist $P_0, P_1, \ldots, P_n$ such that $P \equiv P_0$, $P' \equiv P_n$ and for $0 \le i \le n$, $P \xrightarrow{\tilde{\alpha}_i} P_i$. We denote $P \xrightarrow{\alpha^* \lambda} P'$, where $(\alpha^* \lambda) \in \{\lambda, \tau\, \lambda, \tau\tau\, \lambda, \ldots\}$. We write $P \longrightarrow P'$ if there exists $\tilde{\alpha}$ such that $P \xrightarrow{\tilde{\alpha}} P'$. $P \xrightarrow{\tilde{\alpha}}$ means there exists some P' such that $P \xrightarrow{\tilde{\alpha}} P'$. P is *stable* if P can make no hidden actions, i.e, $P \overset{\tau}{\not\rightarrow}$, and P is an end-state if P can make no action at all .i.e. $P \not\rightarrow$. We denote $\lambda \tilde{\in} \tilde{a}$ to mean that either λ or $\bar{\lambda}$ is among the labels appearing in $\tilde{a}$.

Bibliography

[AJS05] A.E. Abdallah, C.B. Jones, and J.W. Sanders, editors. *Communicating Sequential Processes: The First 25 Years*, LNCS 3525. Springer, 2005.

[Bae05] J.C.M. Baeten. A brief history of process algebra. *Theoretical Computer Science*, 335(2-3):131–146, 2005.

[BG95] J.P. Bowen and M.J.C. Gordon. A Shallow Embedding of Z in HOL. *Information and Software Technology*, 37(5-6):269–276, 1995.

[BGG$^+$92] R.J. Boulton, A. Gordon, M.J.C. Gordon, J. Harrison, J. Herbert, and J.V. Tassel. Experience with Embedding Hardware Description Languages in HOL. In V. Stavridou, T.F. Melham, and R.T. Boute, editors, *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design*, pages 129–156. North-Holland, 1992.

[BK89] J.A. Bergstra and J.W. Klop. ACP$_\tau$: A universal axiom system for process specification. In M. Wirsing and J.A. Bergstra, editors, *Algebraic Methods: Theory, tools and applications*, LNCS 394, pages 447–463. Springer, 1989.

[Box97] D. Box. *Essential COM.* Addison–Wesley, 1997.

[BPS01] J.A. Bergstra, A. Ponse, and S.A. Smolka. *Handbook of Process Algebra.* Elsevier, 2001.

[Bry92] R.E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[BW99] S. Berghofer and M. Wenzel. Inductive Datatypes in HOL - Lessons Learned in Formal-Logic Engineering. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *TPHOLs '99*, LNCS 1690, pages 19–36. Springer, 1999.

[Cam90] A.J. Camilleri. Mechanizing CSP Trace Theory in Higher Order Logic. *IEEE Transactions on Software Engineering*, 16(9):993–1004, 1990.

[Cam91] A.J. Camilleri. A Higher Order Logic mechanization of the CSP Failure-Divergence Semantics. In G. Birtwistle, editor, *IV Higher Order Workshop*, LNCS 575, pages 123–150. Springer, 1991.

[CCGR99] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: A New Symbolic Model Verifier. In N. Halbwachs and D. Peled, editors, *CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification*, LNCS 1633, pages 495–499. Springer, 1999.

[CGP99] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking.* MIT Press, 1999.

[Doc06] Isabelle Documents. *ISabelle/HOL-Complex- Higher-Order Logic with Complex Numbers*, Accessed December 2006. TU München, October 2005, (http://isabelle.in.tum.de/library/HOL/HOL-Complex/document.pdf).

[DS97] B. Dutertre and S. Schneider. Using a PVS Embedding of CSP to Verify Authentication Protocols. In E.L. Gunter and A.P. Felty, editors, *TPHOLs '97*, LNCS 1275, pages 121–136. Springer, 1997.

[FHRR04] C. Fournet, C.A.R. Hoare, S.K. Rajamani, and J. Rehof. Stuck-free conformance. In R. Alur and D. Peled, editors, *CAV '04*, LNCS 3114, pages 242–254. Springer, 2004.

[Fid] C. Fidge. A Comparative Introduction to CSP, CCS and LOTOS. Technical Report Technical Report 93-24 April 1994, The University of Queensland, Australia.

[Gor00] M. Gordon. From LCF to HOL: a short history. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction: Essays in honour of Robin Milner*, pages 169–185. MIT Press, 2000.

[Gro08] Object Management Group. CORBA: Common Object Request Broker Architecture, Accessed January 2008. (http://www.corba.org/).

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[Hoa06] C.A.R. Hoare. Why ever CSP? *Electronic Notes in Theoretical Computer Science*, 162:209–215, September 2006.

[Hol97] G.J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[IR05] Y. Isobe and M. Roggenbach. A Generic Theorem Prover of CSP Refinement. In N. Halbwachs and L. D. Zuck, editors, *TACAS '05*, LNCS 3440, pages 108–123. Springer, 2005.

[IR07a] Y. Isobe and M. Roggenbach. User guide CSP-Prover Ver 4.0, Accessed July 2007. (http://staff.aist.go.jp/y-isobe/CSP-Prover/CSP-Prover.html).

[IR07b] Y. Isobe and M. Roggenbach. Webpage on CSP-Prover, Accessed April 2007. (http://staff.aist.go.jp/y-isobe/CSP-Prover/CSP-Prover.html).

[Kam07] F. Kammueller. CSP Revisited. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics*, LNCS 4732, pages 144–154, September 2007.

[Lim07] F. S. E. Limited. Webpage on FDR2: Failures-Divergence Refinement, Accessed July 2007. (http://www.fsel.com/).

[LO01] K. Lau and M. Ornaghi. A formal approach to software component specification. In D. Giannakopoulou, G.T. Leavens, and M. Sitaraman, editors, *Proceedings of Specification and Verification of Component-based Systems Workshop at OOPSLA*, pages 88–96. IEEE Computer Society Press, Oct 2001.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[Mil99] R. Milner. *Communicating and Mobile systems: the π-calculus.* Cambridge University Press, 1999.

[Nes92] M. Nesi. A Formalization of the Process Algebra CCS in Higher Order Logic. Technical Report UCAM-CL-TR-278, University of Cambridge, Computer Laboratory, December 1992.

[NPW02] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic.* LNCS 2283. Springer, 2002.

[NPW06] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle's Logic: HOL* , Accessed December 2006. TU München, October 2005 (isabelle.in.tum.de/doc/logics-HOL.pdf).

[ORS92] S. Owre, J.M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction*, LNCS 607, pages 748–752. Springer, 1992.

[Pau98] L.C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1-2):85–128, 1998.

[RBH81] A.W. Roscoe, S.D. Brookes, and C.A.R. Hoare. A Theory of Communicating Sequential Processes. Technical Report PRG-16, Oxford University Computing Laboratory, May 1981.

[Ros98] A.W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1998.

[Ros06] A.W. Roscoe. Private conversation with M. Roggenbach, 2006.

[Ros07] A.W. Roscoe. Revivals, Stuckness and the hierarchy of CSP Models. (Revision of 2005 draft) http://web.comlab.ox.ac.uk/oucl/work/bill.roscoe/publications/105b.pdf, Accessed December 2007.

[RRS04] A.W. Roscoe, J.N. Reed, and J.E. Sinclair. Responsiveness of Interoperating Components. *Formal Aspects of Computing*, 16:394–411, 2004.

[RRS06] J.N. Reed, A.W. Roscoe, and J. Sinclair. Machine-verifiable responsiveness. *Electronic Notes of Theortical Computer Science*, 145:185–200, 2006.

[RSG$^+$01] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and A.W. Roscoe. *Modelling and Analysis of Security Protocols.* Addison–Wesley, 2001.

[SIR07] D. G. Samuel, Y. Isobe, and M. Roggenbach. Reasoning on Responsiveness – Extending CSP-Prover by the Model R. In Einar Broch Johnson, Olaf Owe, and Gerardo Schneider, editors, *NWPT'07 / FLACOS'07*, Reseach Report 366, pages 61–63. Universitetet i Oslo, Institutt for informatikk, Norway, 2007.

[SIRar] G. Samuel, Y. Isobe, and M. Roggenbach. The stable revival model in csp-prover. In *AVOCS08*, To appear.

[Szy02] C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison–Wesley, 2002.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[Tha99] T.L. Thai. *Learning DCOM.* O'Reilly & Associates, 1999.

[TW97] H. Tej and B. Wolff. A Corrected Failure Divergence Model for CSP in Isabelle/Hol. In J. S. Fitzgerald, C. B. Jones, and P. Lucas, editors, *FME '97*, LNCS 1313, pages 318–337. Springer, 1997.

[UKG08] Homepage of the Grand Challenges, Accessed March 2008. Dependable systems evolution: A grand challenge for computer science (http://www.nesc.ac.uk/esi/events/Grand_Challenges/proposals/dse.pdf).

[UML08] Object Management Group UML. Unified Modeling Language, Accessed January 2008. (http://www.uml.org/).

[Wen06] M. Wenzel. The Isabelle/Isar Reference Manual, Accessed December 2006. Part of the Isabelle distribution. TU München, October 2005 (http://isabelle.in.tum.de/dist/Isabelle/doc/isar-ref.pdf).

[WH05] K. Wei and J. Heather. Embedding the Stable Failures Model of CSP in PVS. In J. Romijn, G. Smith, and J. van de Pol, editors, *Proceedings of Fifth International Conference on Integrated Formal Methods*, LNCS 3771, pages 246–265. Springer, 2005.