Communicating Multi-Stack Visibly Pushdown Processes

by

Davidson Madudu

A thesis submitted to the
Department of Computer Science
in conformity with the requirements for
the degree of Master of Science

Bishop's University
Sherbrooke, Quebec, Canada
March 2016

Abstract

Visibly Pushdown Languages (VPL) were proposed as a formalism to model and verify complex, concurrent and recursive computing systems. However, the lack of closure under shuffle for VPL makes it unsuitable for the specification and verification of such complex systems. Multi-stack Visibly Pushdown Languages (MVPL) appear to give a more realistic expression of concurrency and recursion in computational systems, but they turn out to have similar limitations. However, natural modifications of the definition of MVPL operations result in a formalism that becomes suitable for the specification and verification of complex computing systems.

With this result in mind, we introduce an MVPL-based process algebra called Communicating Multi-stack Visibly Pushdown Processes (CMVP). CMVP defines a superset of CSP by combining the interesting properties of finite-state algebras (such as CSP) with the context-free features of MVPL. Unlike any other process algebra, CMVP includes support for parallel composition but also for the general form of recursion. We present the syntax, operational semantics, trace semantics, trace specification, and trace verification of CMVP. In addition to the above, a CMVP trace observer can extract stack and module information from a trace; as a result one can specify and verify many software properties which cannot be specified in other existing process algebra. Such properties include the access control of a module, stack limits, concurrent stack properties, internal property of a module, pre- and post-conditions of a module, etc. CMVP lays the basis of algebraic conformance testing for infinite-state processes, such as application software.

Acknowledgments

Firstly, it is great gratitude that I acknowledge God's grace which was clearly at work from the inception of this dissertation and up until this moment.

I would also like to express my sincere gratitude to my supervisor, Prof. S. Bruda for his continuous support, patience, motivation and guidance throughout my MSc. program. His immense knowledge and guidance was of great help all through my degree program and with the completion of this dissertation.

Besides my supervisor, I would like to acknowledge and thank the members of my thesis defence committee: Prof. L. Bentabet, Prof. J. Dingel, and Prof. L. Jensen, for letting my defence be an enjoyable experience and for their insightful suggestions, comments and encouragement.

I also want acknowledge the support and encouragement of my parents and my sister, whose actions and words pulled me up when I faced what seemed to be a mountain.

I would also like to thank the Bishop's University Computer Science department for providing me with a platform and the necessary support needed for me to attain goals I had set for myself.

Last but not the least, I duly acknowledge all my friends and course mates who provided a helping hand when I was in need. To all I say a big thank you.

Contents

1 Introduction **1**
- 1.1 Process Algebra . . . 2
- 1.2 Visibly Pushdown Languages . . . 3
- 1.3 Multi-Stack Visibly Pushdown Languages . . . 4
- 1.4 Thesis . . . 5
- 1.5 Dissertation Summary . . . 6

2 Preliminaries **8**
- 2.1 Labelled Transition System . . . 8
- 2.2 Communicating Sequential Processes . . . 9
- 2.3 Visibly Pushdown Automata . . . 10
- 2.4 Multi-stack Visibly Pushdown Automata . . . 12
- 2.5 Trace Semantics . . . 15
  - 2.5.1 Traces . . . 16
  - 2.5.2 Trace Semantics . . . 17
  - 2.5.3 Specification with Traces . . . 18
  - 2.5.4 Verification with Traces . . . 19
- 2.6 Previous Work . . . 20

3 Communicating Multi-Stack Visibly pushdown Processes **23**
- 3.1 The Operational Semantics of CMVP . . . 25
  - 3.1.1 Prefix Choice . . . 27
  - 3.1.2 Internal Event . . . 28
  - 3.1.3 Choice . . . 28
  - 3.1.4 Recursion . . . 30
  - 3.1.5 Parallel Composition . . . 31
  - 3.1.6 Hiding . . . 33
  - 3.1.7 Abstract . . . 33
  - 3.1.8 Renaming . . . 35
  - 3.1.9 Sequential Composition and Interrupt . . . 36
- 3.2 CMVP Is a Process Algebra . . . 38

4 A Detailed Example **41**
4.1 Prefix Choice . . . 42
4.2 Internal Event . . . 42
4.3 Choice . . . 43
4.4 Recursion . . . 44
4.5 Parallel Composition . . . 44
4.6 Hiding . . . 45
4.7 Abstract . . . 45
4.8 Renaming . . . 46
4.9 Sequential Composition and Interrupt . . . 47

5 CMVP Trace Semantics **48**
5.1 Prefix Choice . . . 48
5.2 External Choice . . . 49
5.3 Internal Choice . . . 50
5.4 Parallel Composition . . . 51
5.5 Hiding . . . 53
5.6 Renaming . . . 54
5.7 Sequential Composition . . . 56
5.8 Interrupt . . . 56
5.9 Recursion . . . 58
5.10 Abstract . . . 59

6 Trace Specification and Verification in CMVP **61**
6.1 CMVP Trace Functions . . . 61
6.1.1 Abstract Function . . . 61
6.1.2 Stack Extract . . . 62
6.1.3 Module Extract . . . 62
6.1.4 Completeness . . . 62
6.2 CMVP Trace Specification . . . 63
6.2.1 Access Control . . . 63
6.2.2 Stack Limit . . . 63
6.2.3 Concurrent Stack Properties . . . 63
6.2.4 Internal Properties of a Module . . . 64
6.2.5 Pre- and Post-Conditions . . . 64
6.3 CMVP Trace Verification . . . 65
6.3.1 Prefix Choice . . . 65
6.3.2 Choice . . . 65
6.3.3 Parallel Composition . . . 67
6.3.4 Hiding . . . 67
6.3.5 Abstract . . . 68
6.3.6 Renaming . . . 68
6.3.7 Sequential Composition . . . 69

6.3.8 Interrupt 69
6.3.9 Recursion 69

7 Conclusions **70**
7.1 Advantages of CVP over Other Process Algebrae 71
7.2 Future 72

Bibliography **74**

List of Figures

2.1 Operational Semantics of CSP [17] 11

3.1 Prefix choice 27
3.2 Internal action (a), internal choice (b) 28
3.3 External choice 29
3.4 Recursion 29
3.5 Alphabetized parallel 31
3.6 Hiding 33
3.7 Abstract 34
3.8 Forward renaming 35
3.9 Backward renaming 36
3.10 Sequential composition 36
3.11 Interrupt 37

4.1 Illustration of the possible transitions of state $P5$ of process P_ϵ 42
4.2 Illustration of internal choice: $P \sqcap Q$ 43
4.3 A run of the parallel composition between processes P_ϵ and Q_ϵ 45
4.4 Illustration of the possible transitions of state $P5 \setminus \{c\}$ 46
4.5 An abstracted configuration and an un-abstracted configuration of P_ϵ . . . 46
4.6 A run illustrating a sequential composition between P_ϵ and Q_ϵ 47
4.7 A run illustrating an interrupt between P_ϵ and Q_ϵ 47

5.1 Laws for external choice 50
5.2 Law for internal choice 51
5.3 Laws for alphabetized parallel 52
5.4 Laws for hiding 53
5.5 Laws for Renaming 55
5.6 Laws for sequential composition 57
5.7 Laws for interrupt 57

Chapter 1

Introduction

In today's world, there is a heavy reliance on sophisticated computing systems. For instance, our financial, transportation, government and medical sectors are all managed by using various computing infrastructures. Many of these systems perform in real-time and directly impact individuals. As a result of this heavy dependence, it must be ensured that these systems comply with their specifications. This is archived by carrying out verification and validation tests on the computing systems before they are deployed, and by so doing the chances of failures are minimized. Various tools have been developed for the specification and verification of computing systems.

Formal methods in particular have been hugely successful in proving the correctness of computing systems since their introduction over two decades ago. Formal methods are able to provide a mathematical guarantee of correctness, much needed for any mission-critical computing system.

This being said, the modelling of recursive and concurrent systems is a major challenge in the software verification arena. Various non regular properties need to be modelled to create a specification and to perform verification on these complex systems. Current standard verification techniques such as model checking [13] or finite state process algebrae [4] are unable to model regular properties and so are difficult to use for complex application software. Context-free techniques such as basic process algebra or BPA [6], etc. present a

number of issues stemming from the lack of closure of context-free languages under several operations, which in turn limit their compositionality. This has all lead to wide spread research in formal methods to develop a concurrent process algebra that would easily specify and verify concurrent and recursive systems.

1.1 Process Algebra

A process algebra is a mathematical framework used to model a computational system in other to help analyze the behaviour of that system [27]. In carrying out this analysis process algebras rely on both equational logic and non-equational logic [27]. Pioneering research on process algebra was carried out by both Milner [22] and Hoare [17].

Today, we have a fairly large number of process algebraic theories developed by various researcher for modelling various aspects of computational systems. Examples of process algebras in use today are: Communicating Sequential Processes or CSP [8, 24], Calculus of Communicating Systems or CCS [22] and Algebra of Communicating Processes or ACP [5]. In general, process algebra theories are defined by three semantic approaches, namely: operational semantics, denotational semantics and axiomatic semantics.

Operational semantics models the behaviour of a system as an execution of an abstract machine consisting only of a set of states and a set of transitions [7, 21]. In comparison to operational semantics, *denotational semantics* is a more abstract semantic approach. It models the behaviour of a system by a function transforming input into output [25]. By using this semantical approach, behavioural equivalences such as refinement ordering and congruence can be introduced to check equivalence between related systems which have system behaviours that are indistinguishable to an external observer. Finally, the *axiomatic semantics* checks system behaviour by using axiomatic proof methods to validate the correctness of a system against a proposed specification [14, 16].

Process algebras adopt different kinds of denotational and axiomatic approaches for the specification and verification of system behaviour. For instance, CCS has been studied

under bisimulation and testing semantics [22], CSP under trace and failure semantics (a variants of testing semantics) [8, 24], and ACP under bisimulation and branching bisimulation semantics [5]. However, in all cases one needs to first establish an operational semantics.

Generally, systems consist of several levels of sub-systems. Using the congruence and refinement relations provided by a process algebra one can determine if these various sub-systems conform to one another. These relations are typically substitutive, meaning that related sub-systems may be used interchangeably inside a larger system. As a result, compositional system verification can be carried out since we are able to carry out specification and verification on sub-systems in isolation from their parent system.

1.2 Visibly Pushdown Languages

Visibly pushdown languages (VPL) [3] lay in the spectrum between balanced languages and deterministic context-free languages [3]. They were introduced as a possible basis for formal verification [3]. VPL share many of the interesting properties that the regular languages have. Their nondeterministic acceptors are equally as expressive as their deterministic counterparts. They have closure under union, intersection, complementation, concatenation, Kleene star, prefix, and language homomorphisms; however, they lack closure under shuffle [11]. Membership, emptiness, language inclusion, and language equivalence are all decidable for VPL.

VPL are accepted by *visibly pushdown automata (vPDA)*, which are push-down automata with stack behaviour controlled by the input alphabet. A vPDA operates over an alphabet that is partitioned into three disjoint sets of call, return, and local symbols. Any input symbol can change the control state, but only calls and returns can modify the stack content. While accepting a call symbol a vPDA must push one symbol on the stack, and while accepting a return symbol it must pop one symbol from the stack (unless the stack is empty). vPDAs model the execution of a recursive module naturally by using call symbols

to represent the invocation of modules, return symbols to represent the returns from modules, and local symbols for all the other actions. Attempts have then been made to develop a VPL-based concurrent process algebra [27], however the lack of closure under shuffle has prevented the said efforts [11].

1.3 Multi-Stack Visibly Pushdown Languages

Multi-stack visibly pushdown languages (MVPL) [19] is a natural extension of VPL. MVPL lay in the spectrum beyond context-free languages. MVPL share the same interesting properties that VPL have, however, they also have similar limitations. Just like VPL, they also lack closure under shuffle [11].

MVPL are accepted by *multi-stack visibly pushdown automata (MVPDA)*, which operate on n stacks for some $n \geq 1$. The input alphabet also control stack behaviour. A MVPDA operates over an n-stack call-return alphabet partitioned into $n + 1$ pair-wise disjoint sets of alphabets (n pairs of call and returns alphabets plus one alphabet of local symbols). Similarly to VPL, an input symbol can change the control state, but only calls and returns can modify the content of the stacks. As before, while executing a call symbol associated to the i-th stack a MVPDA must push one symbol on that stack, and while executing a return symbol associated to the i-th stack it must pop one symbol from that stack (unless the i-th stack is empty). MVPDAs naturally capture the properties of recursive modules by representing the invocation of a module by a call symbol, the return from a module by a return symbol, and all the other actions by local symbols. This time however the modules can operate in n separate threads of execution, while each such a thread has one associated stack.

The original operations over MVPL were very restricted, in the sense that the usual operations over two MVPL (such as union, concatenation, etc.) are only defined when the two languages are over exactly the same n-stack call-return alphabet. Such a restriction was later relaxed [11] to the restriction that two MVPL can be composed iff the sets of call,

local, and return symbols of the two languages do not overlap (meaning that a call symbol in one language is not a return or a local symbol in the other, and so on).

However, this unrestricted MVPL lacks any useful closure property. Such properties were restored by the introduction of a natural stack renaming process (discussed in section 2.4), which not only imitate what happens in real life (namely, the concurrent execution of threads), but also restores all the closure properties of MVPL, with closure under shuffle on top [11]. Indeed, based on this stack renaming process, one can easily define "disjoint" variants of all the interesting operators such that MVPL are closed under all of them [11]. This effort paves the way for an MVPL-based compositional specification and verification of complex systems. Hence, we adopt the model of disjoint operations over the unrestricted variant of MVPL and thus use MVPL as the underlying model of CMVP.

1.4 Thesis

Classical process algebras such as CCS, ACP and CSP are only able to specify regular properties, since regular languages are used as their domain languages. Regular languages are closed under all the operations that are required to create a process algebra namely, union, Kleene star, intersection, shuffle, hiding, renaming, concatenation, and prefix. Since regular languages are unable to specify non-regular properties they are however unable to adequately model recursive and concurrent behaviours in computational systems. Unlike regular languages, context-free languages are able to capture both regular and non regular properties of computing systems. However, they lack closure under the critical operation of intersection.

On the other hand, both VPL and MVPL naturally capture the properties of recursive and concurrent systems [11]. However, they both have identical limitations; the lack of closure under shuffle effectively prevents both VPL-based and MVPL-based compositional approaches to the specification of concurrent and recursive systems [11]. The recent introduction of an unrestricted MVPL variant and a natural stack renaming process (discussed

in Section 2.4) allows us to use the defined set of disjoint operation by which they may be operated on MVPL and effectively creates the potential for an MVPL-based concurrent process algebra [11].

Our thesis is therefore that a fully compositional concurrent MVPDA-based process algebra is possible. We are thus presenting a process algebra called Communicating Multi-stack Visibly Pushdown Process (CMVP). We also present the operational semantics and the trace model of CMVP, using the semantical approach of CSP.

1.5 Dissertation Summary

This dissertation uses multi-stack pushdown languages (MVPL) [19] to develop a fully compositional process algebra that will naturally model recursive properties and concurrency in computational systems. Chapter 2 begins by presenting preliminary information. Section 2.1 establishes a *labelled transition system (LTS)* semantics for MVPDA, the underlying semantic model for all the process algebras. The underlying LTS of a MVPDA-based process algebra is an infinite-state machine. Every state of such an LTS is represented by the combination of a MVPDA state and the current stack content of the stack in operation associated to that state. Special attention is given to CSP in Section 2.2 since our process algebra closely follows the semantic approach of CSP. Sections 2.3, 2.4, and 2.5 then introduce formally vPDA, MVPDA, and traces semantics, respectively. Chapter 2 concludes with a summary of related work.

In Chapter 3 the CMVP syntax is first introduced. We then show how the operators of a concurrent process algebra along with a new operator *abstract* can be applied in the MVPL setting. We apply our technique on the operators of CSP [7, 8, 15, 17, 24] (a random choice, our formalism works with all the other finite-state process algebras). We thus introduce a MVPDA-based process algebra called Communicating Multi-stack Visibly Pushdown Processes (CMVP) as a superset of CSP; when all the input symbols are locals then CMVP is equivalent to CSP. In Section 3.1 we describe the operational semantics of

CMVP. Then Theorem 3.2.1 (Section 3.2) shows that CMVP is indeed an algebra, being closed under all its operations. In all, we introduce the syntax and the structural operational semantics of CMVP.

We carry out a detailed example in Chapter 4, that shows in detail the MVPDA model for the various constructs of CMVP. We provide a clearer illustrations by means of examples of all the CMVP operations.

A trace model for CMVP in discussed in Chapters 5. The trace semantics for CMVP is presented first. Then four functions that support our semantical model are defined on CMVP traces: abstract $\mathfrak{A}$, stack extract $\mathfrak{S}$, module extract $\mathfrak{M}$, and completeness $\mathfrak{C}$.

A trace framework for CMVP trace specification and verification is then presented in Chapter 6. In particular we demonstrate in Section 6.2 that useful properties for software verification (which context-free or regular process algebras are unable to specify) can be specified in CMVP. We then proceed to provide the trace proof system for CMVP, used to verify the properties defined in Section 6.2.

Chapter 7 brings a conclusion to the dissertation by enumerating CMVP's advantage over other process algebras and further research that could be done to advance this study.

Chapter 2

Preliminaries

2.1 Labelled Transition System

A labelled transition system (LTS) [9] is a tuple $(\Theta, \Sigma, \Delta, I)$, where Θ is a set of states , Σ is a finite set of actions (not containing the internal action τ), $I \in \Theta$ is the initial state, and Δ is the transition relation such that $\Delta \subseteq \Theta \times (\Sigma \cup \{\tau\}) \times \Theta$. If Δ is unambiguous and understood from the context, then we often use the following shorthands: $P \xrightarrow{a} Q$ whenever $(P, a, Q) \in \Delta$, $P \xrightarrow{a}$ whenever there exists a Q such that $P \xrightarrow{a} Q$, and $P \not\xrightarrow{a}$ whenever $P \xrightarrow{a}$ does not hold. Some times one assumes a global set of states, a global set of actions, and a global transition relation for all the labelled transition systems; in this case, a particular labelled transition system is identified solely by its initial state. We therefore blur the difference between state and labelled transition systems as long as the set of states, the set of actions, and the transition relation are all understood from the context.

A run of a labelled transition system M is a sequence $q_0 \tau q_{01} \tau \cdots \tau q_{0m_0} a_1 q_1 \tau q_{11} \tau \cdots \tau q_{1m_1} a_2 q_2 \cdots a_k q_k \tau q_{k1} \tau \cdots \tau q_{km_k}$ such that $q_0 = I$, $q_{j-1i} \xrightarrow{\tau} q_{ji}$ for all $1 \leq i \leq k$, $1 \leq j \leq m_i$, and $q_{i-1m_{i-1}} \xrightarrow{a_i} q_i$ for all $1 \leq i \leq k$. The trace of this run is the sequence $a_1 a_2 \cdots a_k$. The run is maximal whenever there is no x such that $q_{km_k} \xrightarrow{x}$. The trace of a maximal run is called a complete trace. The language traces(M) contains exactly all the traces of all the possible runs of M. Similarly, ctraces(M) contains exactly all the complete traces of all the possible maximal runs of M.

The weakest notion of equivalence between labelled transition systems is trace equivalence: two labelled transition systems are equivalent if their sets of traces are identical. By contrast, the largest (or finest) notion of equivalence between labelled transition systems is the notion of bisimilarity [10]. Two bisimilar transition systems have not only the same set of traces, but their internal structure is identical: Given a global set of states Θ, a global set of actions Σ, and a global transition relation $\rightarrow$, a binary relation $\sim$ over labelled transition systems is a bisimulation if for every pair of states p and q such that $p \sim q$ and for every action $a \in \Sigma$:

1. $p \xrightarrow{a} p'$ implies that there is a q' such that $q \xrightarrow{a} q'$ and $p' \sim q'$; and symmetrically
2. $q \xrightarrow{a} q'$ implies that there is a p' such that $p \xrightarrow{a} p'$ and $p' \sim q'$.

Several additional equivalence relations between LTS exist. Their power of discrimination lies between trace equivalence and bisimilarity [10].

2.2 Communicating Sequential Processes

Special attention is given to Communicating Sequential Processes (CSP), because our research uses the semantic approach used in CSP as its model. CSP uses trace and failure semantics as its denotational and axiomatic semantic [8, 24]. It models a system's behaviour or processes using eight major operators: event prefix, choice, recursion, parallel composition, hiding, renaming, sequential composition, and interrupt [5, 7, 8, 15, 17, 22, 23, 24]. A prefix or suffix of a process is also regarded as a process, therefore the domain language is closed under prefix and suffix. CSP uses labelled transition systems (LTS) as its underlying semantic model.

The syntax of CSP is defined as follows:

$$S \quad ::= \quad x : A \rightarrow S(x) \mid S \,\Box\, R \mid S \sqcap R \mid X \mid S \,{}_A\|_B R \mid S \setminus A \mid f(S) \mid f^{-1}(S) \mid S; R \mid S \triangle R$$

where Σ is the set of (elementary) actions, S and R range over CSP processes, x ranges over Σ, A and B range over 2^{Σ}, and f ranges over the set $\{f : \Sigma \rightarrow \Sigma : \forall a \in \Sigma\colon\ f^{-1}(a)$ is

finite $\wedge$ $f(a) = \checkmark$ iff $a = \checkmark\}$ of Σ-transformations. The CSP prefix choice $x : A \to S(x)$ is a process which may engage any $x \in A$ and then its behaviour depends on that choice. $S \sqcap R$ denotes a process which may behave as either S or R independently of its environment. $S \,\Box\, R$ denotes a process which may behave as either S or R, the choice being influenced by the environment, provided that such influence is exerted on the first occurrence of an external event of the composite process. ${S}_{A}\|_{B}R$ denotes a process which behaves like the alphabetized parallel composition of S and R, with the following meaning: any external event performed by the composition must lie in $A \cup B$; the composition may then perform an event a only if $a \in A \setminus B$ and S may perform a, or $a \in B \setminus A$ and R may perform a, or $a \in A \cap B$ and both S and R may perform (synchronously) a. $S; R$ denotes the sequential composition of S followed by R and $S \setminus A$ is the process which behaves like S except that all the occurrences of $a \in A$ are rendered invisible to the environment. The processes $f(S)$ and $f^{-1}(S)$ derive their behaviour from that of S in that if S may perform the event a then $f(S)$ may perform $f(a)$ while $f^{-1}(S)$ may engage in any event b such that $f(b) = a$. $S \triangle R$ denotes R interrupting the process S: R may begin execution at any point throughout the execution of S; the performance of the first external event of R is the point at which control passes from S to R and then S is discarded. A process name X may be used as a component process in a process definition. It is bound by the definition $X = S$ where S is an arbitrary process which may include process name X.

The most common underlying semantical model of CSP (like any other process algebra) is LTS. One way of presenting the operational semantics of CSP using LTS is by using structural operational semantics rules as shown in Figure 2.1, with $A^{\checkmark} = A \cup \{\checkmark\}$.

2.3 Visibly Pushdown Automata

In what follows ε is used to denote the empty string and only the empty string.

A visibly pushdown automaton (VPDA) [3] is a tuple $M = (\Phi, \Phi_{in}, \widetilde{\Sigma}, \Gamma, \Omega, \Phi_F)$, where Φ represents a finite set of states, $\Phi_{in} \subseteq \Phi$ is a set of initial states, $\Phi_F \subseteq \Phi$ is the set of final

$$\frac{}{P \xrightarrow{\tau} Q} \qquad \frac{}{(x : A \to P(x)) \xrightarrow{a} P(a)} \; [a \in A]$$

$$\frac{}{P \sqcap Q \xrightarrow{\tau} P} \qquad \frac{P \xrightarrow{\mu} P'}{N \xrightarrow{\mu} P'} \; [N = P]$$

$$\frac{}{P \sqcap Q \xrightarrow{\tau} Q} \qquad \frac{P \xrightarrow{a} P' \qquad Q \xrightarrow{a} Q'}{P \,{}_A\|_B Q \xrightarrow{a} P' \,{}_A\|_B Q'} \; [a \in A^{\checkmark} \cap B^{\checkmark}]$$

$$\frac{P \xrightarrow{a} P'}{\begin{array}{l} P \,\Box\, Q \xrightarrow{a} P' \\ Q \,\Box\, P \xrightarrow{a} P' \end{array}} \qquad \frac{P \xrightarrow{\mu} P'}{\begin{array}{l} P \,{}_A\|_B Q \xrightarrow{\mu} P' \,{}_A\|_B Q \\ Q \,{}_B\|_A P \xrightarrow{\mu} Q \,{}_B\|_A P' \end{array}} \; [\mu \in A \cup \{\tau\} \setminus B]$$

$$\frac{P \xrightarrow{\tau} P'}{\begin{array}{l} P \,\Box\, Q \xrightarrow{\tau} P' \,\Box\, Q \\ Q \,\Box\, P \xrightarrow{\tau} Q \,\Box\, P' \end{array}} \qquad \frac{P \xrightarrow{\mu} P'}{P \setminus A \xrightarrow{\mu} P' \setminus A} \; [\mu \notin A]$$

$$\frac{P \xrightarrow{\checkmark} P'}{P; Q \xrightarrow{\tau} Q} \qquad \frac{P \xrightarrow{\mu} P'}{P \triangle Q \xrightarrow{\mu} P' \triangle Q} \; [\mu \neq \checkmark]$$

$$\frac{P \xrightarrow{f(a)} P'}{f^{-1}(P) \xrightarrow{a} f^{-1}(P')} \qquad \frac{P \xrightarrow{a} P'}{P \setminus A \xrightarrow{\tau} P' \setminus A} \; [a \in A]$$

$$\frac{P \xrightarrow{\checkmark} P'}{P \triangle Q \xrightarrow{\checkmark} P'} \qquad \frac{P \xrightarrow{\mu} P'}{f(P) \xrightarrow{\mu} f(P')} \; [\mu \in \{\tau \cup \checkmark\}]$$

$$\frac{Q \xrightarrow{\tau} Q'}{P \triangle Q \xrightarrow{\tau} P \triangle Q'} \qquad \frac{P \xrightarrow{\mu} P'}{P; Q \xrightarrow{\mu} P'; Q} \; [\mu \neq \checkmark]$$

$$\frac{P \xrightarrow{a} P'}{f(P) \xrightarrow{f(a)} f(P')} \qquad \frac{P \xrightarrow{\mu} P'}{f^{-1}(P) \xrightarrow{\mu} f^{-1}(P')} \; [\mu \in \{\tau \cup \checkmark\}]$$

$$\frac{Q \xrightarrow{a} Q'}{P \triangle Q \xrightarrow{a} Q'}$$

Figure 2.1: Operational Semantics of CSP [17]

states, Γ is the (finite) stack alphabet containing a special bottom-of-stack symbol $\perp$, and Ω is the transition relation, $\Omega \subseteq (\Phi \times \Gamma^*) \times \widetilde{\Sigma} \times (\Phi \times \Gamma^*)$. In addition, $\widetilde{\Sigma} = \{\Sigma_l \cup \Sigma_c \cup \Sigma_r\}$ is a finite set of visibly pushdown input symbols where Σ_l represents the set of local symbols, Σ_c is the set of call symbols and Σ_r is the set of return symbols. $(\Sigma_l, \Sigma_c, \Sigma_r)$ is a partition over $\widetilde{\Sigma}$ (meaning that these three sets are mutually disjoint and also that $\widetilde{\Sigma} = \Sigma_l \uplus \Sigma_c \uplus \Sigma_r$).

Each tuple $((P, \gamma), a, (Q, \delta)) \in \Omega$ (also written $(P, \gamma) \xrightarrow{a} (Q, \delta) \in \Omega$) must have the following form: if $a \in \Sigma_l \cup \{\varepsilon\}$ then $\gamma = \delta = \varepsilon$, else if $a \in \Sigma_c$ then $\gamma = \varepsilon$ and $\delta = a$ (where a is the stack symbol pushed for a), else if $a \in \Sigma_r$ then if $\gamma = \perp$ then $\gamma = \delta$ (hence visibly pushdown automata allow unmatched return symbols) else $\gamma = a$ and $\delta = \varepsilon$ (where a is the stack symbol popped for a). In other words, a local symbol is not allowed to modify the stack, while a call always pushes one symbol on the stack. Similarly, a return symbol always pops one symbol off the stack, except when the stack is already empty. Note in particular that ε-transitions (that is, transitions that do not consume any input) are allowed but are not permitted to modify the stack [3].

The notion of run, acceptance, and language accepted by a visibly pushdown automaton are defined as usual: A run of a visibly pushdown automaton M on some string $w = a_1 a_2 \dots a_k$ is a sequence of configurations $(q_0, \gamma_0)(q_{01}, \gamma_0) \cdots (q_{0m_0}, \gamma_0)(q_1, \gamma_1)(q_{11}, \gamma_1) \cdots (q_{1m_1}, \gamma_1)(q_2, \gamma_2) \cdots (q_k, \gamma_k)(q_{k1}, \gamma_k) \cdots (q_{km_k}, \gamma_k)$ such that $\gamma_0 = \perp$, $q_0 \in \Phi_{in}$, $(q_{j-1i}, \varepsilon) \xrightarrow{\varepsilon} (q_{ji}, \varepsilon) \in \Omega$ for all $1 \leq i \leq k$, $1 \leq j \leq m_i$, and $(q_{i-1m_{i-1}}\gamma'_{i-1}) \xrightarrow{a_i} (q_i, \gamma'_i) \in \Omega$ for every $1 \leq i \leq k$ and for some prefixes γ'_{i-1} and γ'_i of γ_{i-1} and γ_i, respectively. Whenever $q_{km_k} \in \Phi_F$ the run is accepting; M accepts w iff there exists an accepting run of M on w. The visibly pushdown language $L(M)$ accepted by M contains exactly all the strings w accepted by M

2.4 Multi-stack Visibly Pushdown Automata

In multi-stack visibly pushdown automaton (MVPL) [19], an n-stack call-return alphabet is a tuple $\widetilde{\Sigma_n} = \{(\Sigma_c^i, \Sigma_r^i)_{1 \leq i \leq n}, \Sigma_l\}$ of pair-wise disjoint alphabets. Σ_c^i represents the finite

set of call symbols of stack i, Σ_r^i represents the finite set of return symbols of stack i, and Σ_l represents the finite set of local symbols. We further use the following notations: $\Sigma_c = \bigcup_{i=1}^{n} \Sigma_c^i$, $\Sigma_r = \bigcup_{i=1}^{n} \Sigma_r^i$, $\Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_l$.

A multi-stack visibly pushdown automaton (MVPDA) [19] over the n-stack call-return alphabet $\widetilde{\Sigma_n} = \{(\Sigma_c^i, \Sigma_r^i)_{1 \le i \le n}, \Sigma_l\}$ is an extension of a VPDA. It is a tuple $M = (\Phi, \Phi_{in}, \widetilde{\Sigma}, \Gamma, \Omega, \Phi_F)$, and just like the VPDA Φ is a finite set of states, $\Phi_{in} \subseteq \Phi$ is a set of initial states, $\Phi_F \subseteq \Phi$ is the set of final states, Γ is the (finite) stack alphabet that contains a special bottom-of-stack symbol $\perp$. The transition relation is Ω and every tuple $((P, \gamma), a, (Q, \delta)) \in \Omega$ (also written $(P, \gamma) \xrightarrow{a} (Q, \delta) \in \Omega$) must have the following form: if $a \in \Sigma_l \cup \{\varepsilon\}$ then $\gamma = \delta = \varepsilon$, else if $a \in \Sigma_c^i$ then $\gamma = \varepsilon$ and $\delta = a$ (where a is the stack symbol pushed for a on the i-th stack), else if $a \in \Sigma_r^i$ then if $\gamma = \perp$ then $\gamma = \delta$ (hence like VPDA, MVPDA allows unmatched return symbols) else $\gamma = a$ and $\delta = \varepsilon$ (where a is the stack symbol popped for a on the i-th stack). Note that transitions that do not consume any input (i.e. ε-transitions) are not allowed to modify the stack. ε-transitions are not permitted in the original MVPDA construction; however, allowing such transitions does not change the language accepted by an MVPDA, so they are introduced for the sake of consistency with the definition of VPDA.

A configuration of M is a tuple (q, γ), where q $\in \Phi$ and $\gamma = (\gamma^1, \ldots, \gamma^n)$ with $\gamma^l \in (\Gamma \backslash \{\perp\})^* \perp$ for all $1 \le l \le n$. For a string $w = a_1 a_2 \ldots a_m \in \Sigma^*$ a run of an MVPDA over w is a sequence of configurations $(q_0, \gamma_0)(q_{01}, \gamma_0) \cdots (q_{0m_0}, \gamma_0)(q_1, \gamma_1)(q_{11}, \gamma_1) \cdots (q_{1m_1}, \gamma_1)$ $(q_2, \gamma_2) \cdots (q_k, \gamma_k)(q_{k1}, \gamma_k) \ldots (q_{km_k}, \gamma_k)$ such that $\gamma_0^l = \perp$ for all $1 \le l \le$ n, $q_0 \in \Phi_{in}$, $(q_{j-1i}, \varepsilon) \xrightarrow{\varepsilon} (q_{ji}, \varepsilon) \in \Omega$ for all $1 \le i \le k$, $1 \le j \le m_i$, whenever $a_i \in \Sigma_c^p \cup \Sigma_r^p, \gamma_{i-1}^l = \gamma_i^l$ for all l $\neq$ p, $(q_{i-1m_{i-1}} \gamma'_{i-1}) \xrightarrow{a_i} (q_i, \gamma'_i) \in \Omega$ for every $1 \le i \le k$ and for some prefixes γ'_{i-1} and γ'_i of γ_{i-1}^p and γ_i^p, respectively; whenever $a_i \in \Sigma_l, (q_{i-1m_{i-1}} \gamma'_{i-1}) \xrightarrow{a_i} (q_i, \gamma'_i) \in \Omega$ and $\gamma_{i-1} = \gamma_i$. Whenever $q_{km_k} \in \Phi_F$ the run is accepting; M accepts w iff there exists an accepting run of M on w. The multi-stack visibly pushdown language (MVPL) $L(M)$ accepted by M contains exactly all the strings w accepted by M.

Disjoint Operations Over MVPL

In the original definition for both VPL [3] and MVPL [19], their operations (complement, union, etc.) are defined only when the two languages have identical alphabets. This allows for closure under most interesting operation, however both languages are not closed under shuffle [11]. To solve this issue [11] relaxes this condition on operations over MVPLs. The result is a set of disjoint operations over MVPL. This however leads to the loss of closure under all the interesting operations [11]. By introducing a natural renaming process as defined below, one is able to restore closure under all the interesting operations and also adds closure under shuffle to the group [11]. We show below the renaming process as defined in [11]:

Let L be an MVPL over the n-stack call-return alphabet $\widetilde{\Sigma_n} = \langle (\Sigma_c^i, \Sigma_r^i)_{1\leq i\leq n}, \Sigma_l\rangle$. The p-stack renaming $\mathcal{R}_p(L)$ of L is an MVPL over the n-stack call-return alphabet $\widetilde{\Sigma'_n} = \langle (\Sigma_c^i, \Sigma_r^i)_{1\leq i\leq n, i\neq p}, (\Sigma_c^{n+1}, \Sigma_r^{n+1}), \Sigma_l\rangle$ such that there exists a bijection $f : \Sigma_c^p \cup \Sigma_r^p \longrightarrow \Sigma_c^{n+1} \cup \Sigma_r^{n+1}$ with $f(x) \in \Sigma_c^{n+1}$ iff $x \in \Sigma_c^p$ and $f(x) \in \Sigma_r^{n+1}$ iff $x \in \Sigma_r^p$. Specifically, $\mathcal{R}_p(L) = \{r(w) : w \in L\}$, where $r : \Sigma \longrightarrow \Sigma'$ is the function $r(x) = x$ for any $x \in \Sigma \backslash (\Sigma_c^p \cup \Sigma_r^p)$ and $r(x) = f(x)$ otherwise, extended as usual to strings by $r(a_1a_2 \ldots a_l) = r(a_1)r(a_2)\ldots r(a_l)$. By abuse of notation $\mathcal{R}_{p_1,p_2,\ldots,p_k}(L) = \mathcal{R}_{p_1}(\mathcal{R}_{p_2}(\ldots \mathcal{R}_{p_k}(L)\ldots))$. Further abusing the terminology we will also use the term stack renaming (or just renaming when there is no ambiguity) for this (composite) renaming.

Some stack renaming $\mathcal{R}_{p_1,p_2,\ldots,p_k}(L)$ of a language L is MVPL iff L is an MVPL. In other words, symbols associated with one stack in a given language can be renamed to the symbols associated with another stack with the following restrictions: if we rename one symbol then we also rename all the other symbols associated with the same stack, no symbol associated with the new stack will be in the language before renaming, and no symbol associated with the old stack will be in the language after renaming. The new stack is always new, meaning that the before-renaming MVPL is not using any symbol from that stack.

2.5 Trace Semantics

Sequences are denoted by listing the elements of the sequence in order of precedence in angled brackets. The set of all the sequences of actions [24] that a process can perform (or that might possibly be recorded) is the set of traces of the process . Hence the empty sequence is denoted by $\langle\rangle$. If we have that A is a set, then A^* will be the set of all finite sequences of elements of A. If we have two sequences, namely; $seq1$ and $seq2$, then the concatenation between these two sequences denoted as $seq1.seq2$ will be the sequence of elements in $seq1$ followed by the sequence of elements in $seq2$. We note that the concatenation operation on sequences is associative. A sequence denoted seq^n describes a concatenation between n copies of the finite sequence seq, hence $seq^0 = \langle\rangle$ (the empty sequence).

If a sequence is not empty, we can perform a prefix operation on it. So if seq is a non-empty sequence then it can be rewritten as $a.seq'$ where a is the prefix (i.e. the first element) of seq and seq' is the suffix (i.e. the remaining elements) of the sequence seq. The following two function can be defined over a sequence $seq = a.seq'$ as above: $head(seq) = a$ and $tail(seq) = seq'$. If $seq = seq''.b$ for some symbol b we define $init(seq) = seq''$ and $foot(seq) = b$. The lenght of a sequence seq denoted by $|seq|$ is the total number of elements it contains. By abuse of notation $a \in seq$ is true iff the symbol a is a member of the sequence seq. We use $\sigma(seq)$ to denote the set of all symbols that are in the sequence seq.

Sequences exhibit various natural relationship with each other. For instance, if we have a sequence $seq2$ such that $seq.seq2 = seq1$, then seq is a prefix of $seq1$, which is denoted as $seq \leq seq1$. Furthermore, $seq \leq_n seq1$ denotes that seq is a prefix of $seq1$ (i.e. $seq \leq seq1$) and their lengths differ by no more than n. Also if $seq \neq seq1$ it will mean that seq is a strict prefix of $seq1$, which can be denoted as $seq < seq1$. The notation $seq \preccurlyeq seq1$ denotes that seq is a (not necessarily contiguous) sub-sequence of $seq1$. $seq \upharpoonright A$ denotes the sub-sequence of all the elements of seq that are in a set A, while the notation $seq \setminus A$ denote the sub-sequence of seq whose elements are not in A. If there is a mapping f on the

elements of seq, then $f(seq)$ is the sequence obtained by applying f to the elements of seq in turn.

2.5.1 Traces

The concept of traces is introduced briefly in Section 2.1 (and further alluded to earlier in this section); we now present this notion in more detail. Traces are used to analyze the interface behaviour of a process (or system) exclusively from an external point of view (or from the view of its environment). A process interact with its environment by performing events or actions in their interface. We note that the environment has no direct access to the internal state of the process or to the internal events that it performs. A very important aspect of process behaviour is the occurrence of events in the right order. The acceptable sequence of actions is stated in the system specification or requirements. Theses requirements will describe constraints on when particular events can occur.

The environment cannot know precisely which internal state a process has reached at any particular point, since it only has access to the projection of the execution onto the interface. To analyze a process with respect to a set requirement, it is necessary to consider those sequences of events that can be observed at the interface of the process. These observations are called traces, and by using traces we can determine equivalence between processes. Therefore, if there are two processes which are indistinguishable at their interfaces, they should be equally appropriate to execute the same specifications. We note that the way both processes are implemented does not have any influence on their respective suitability for the said specification.

If we have a process P, the set of all possible traces of P is denoted as traces(P). Traces are a particular class of finite sequences of events drawn from an alphabet which represents execution. Events in a process's execution cannot occur after termination so any termination event ✓ occurring in a trace must appear at the end. The set of all traces is defined as: $TRACE = \{tr | \sigma(tr) \subseteq \Sigma^{\checkmark} \wedge |tr| \in \mathbb{N} \wedge \checkmark \notin \sigma(init(tr))\}$. Since all traces are

sequences, they inherit all of the sequence operators over sequences. However, sequence concatenation maps traces $tr1$ and $tr2$ to a trace $tr1.tr2$ only if $\checkmark \notin \sigma(tr1)$. Thus tr^n will be a trace only if $\checkmark \notin \sigma(tr)$. If a function f maps Σ to Σ and $f(\checkmark)$ to $\checkmark$, then $f(tr)$ will always be a trace. The notation $P \stackrel{tr}{\Rightarrow} P'$ means there is a sequence of transitions whose initial process is P and whose final process is P' after executing tr. The notation $P \stackrel{tr}{\Rightarrow}$ is shorthand for $\exists P' : P \stackrel{tr}{\Rightarrow} P'$.

2.5.2 Trace Semantics

Trace semantics defines processes directly in terms of their traces, so that their entire analysis is lifted to a more abstract level. Operational characterization is too low level for reasoning about processes, since the level of abstraction remains that of process executions, with traces being one of the consequences of the execution. Since the traces of a composite process depend only on the traces of its components, all of the operators of the language can be understood at this abstract level. This permits a *compositional* semantic model, which allows all processes to be considered only in terms of their sets of traces, and at no point do we considered explicitly the underlying executions of processes.

Trace semantics models each process by associating it with a set of traces. The set of traces of a process is the set of all possible sequences of events that may be observed during some execution of the process. Processes are *trace equivalent* iff they have exactly the same set of possible traces. This form of equivalence is denoted $=_T$, and its definition is that $P =_T Q$ *iff* $\text{traces}(P) = \text{traces}(Q)$. The theory of trace equivalence allows the definition of algebraic laws for individual operators, and also laws concerning the relationships between various operators. By applying these laws we can manipulate process descriptions from one form to another without changing the associated set of traces.

Most of the laws are generally defined with algebraic properties such as associativity and commutativity of operators (which allows the composition of components in any order), idempotence, and the identification of units and zeroes for particular operators (which allows

the simplification of process description). Other laws are defined as relationships between different operators (for instance, the expansion of a parallel composition into a prefix choice process).

STOP is a process which does not execute any event:

$$\text{traces}(STOP) = \{\langle\rangle\}$$

while *SKIP* can execute a termination ✓. The only traces *SKIP* exhibits are the empty trace and the singleton trace containing ✓:

$$\text{traces}(SKIP) = \{\langle\rangle, \langle\checkmark\rangle\}$$

A vital process for the definition of laws in the traces axiomatic model is RUN. It can execute any sequence of events:

$$\text{traces}(RUN) = \{tr | tr \in TRACE\}$$

It is defined recursively as: $RUN = (x : \Sigma \rightarrow RUN) \ \Box \ SKIP$. The process RUN_A is defined to be the process with interface A that can always execte any event in its interface: $\text{traces}(RUN_A) = \{tr | tr \in TRACE \wedge \sigma(tr) \subseteq A\}$.

2.5.3 Specification with Traces

A system specification is a tool used for the comparison of computational systems to check for correctness. When a system is developed, it is designed to satisfy particular requirements. Using trace semantics to model specifications will allow us to judge a system against its given specification. In the trace model, a specification of a process [8, 24] is given in terms of the traces the process may engage in. The model characterizes the traces that a system or process can or should have and those that should not be performed. A process satisfies its specification if all of its executions are acceptable. In other words, regardless of the choices made by the process, it is guaranteed that the process will not violate its specification during any of its executions. If $S(tr)$ is a predicate on trace tr, then process P satisfies

(or meets) $S(tr)$ if $S(tr)$ holds for every traces tr of P: $P \vdash S(tr) = \forall tr \in \text{traces}(P) : S(tr)$. The specification $S(tr)$ is said to be a *property-oriented specification*, because the required property is captured by $S(tr)$ as a restriction on traces. The predicate S may be expressed in any notation, although first order logic and elementary set and sequence notations are generally sufficient.

A process P fails to satisfy a specification $S(tr)$, only because it has some (finite) trace for which S fails to hold: there is a point in its execution where the performance of a particular action takes the execution of P outside the specification $S(tr)$. To satisfy a trace specification, it is necessary to ensure that no violating action occur at any point in an execution. This kind of specification is referred to as a *safety* specification. It stipulates that nothing 'bad' should ever happen, and it is precisely this kind of property that is expressed as specification on traces.

2.5.4 Verification with Traces

Trace semantics [8, 24] have a compositional nature which paves the way for a compositional proof system that can be applied in describing trace specifications. Hence, the specification of a process can be deduced from the specifications of its components, in a way which reflects the trace semantics of the operators. The proof system is defined as a set of proof rules for all of the operators. Each rule provides a specification which holds for a composite process starting from antecedents which describe specifications which hold for the component processes. There are three rules whose validity hold for all processes, due to the nature of the $\vdash$ specification:

The first rule is that any process satisfies the vacuous specification $true(tr)$, which holds for all traces tr:

$$\frac{}{P \ \vdash \ true(tr)}$$

The second rule is that any specification may be weakened:

$$\frac{P \ \vdash \ S(tr)}{P \ \vdash \ T(tr)} \ \left[\forall \ tr : TRACE : S(tr) \Rightarrow T(tr)\right]$$

The final rule states that if $S(tr)$ and $T(tr)$ have been established separately, then the specification consisting of their conjunction is also established

$$\frac{P \vdash S(tr) \quad P \vdash T(tr)}{P \vdash (S \wedge T)(tr)}.$$

Process $STOP$ has only one trace: the empty trace. The strongest specification that process $STOP$ can satisfy is that $tr = \langle\rangle$. This is encapsulated in the rule:

$$\frac{}{STOP \vdash tr = \langle\rangle}$$

The rule has no antecedents, corresponding to the fact that $STOP$ has no component processes. The weak rule given above can be used to show that any specification which is satisfied by any process must be satisfied by $STOP$.

Process $SKIP$ does nothing except terminate successfully. There are only two possible traces, one for the situation before it has terminated successfully, and the other for the situation where termination occurs. These two traces are $\langle\rangle$ and $\langle\checkmark\rangle$,so the inference rule, which has no antecedents, is as follows:

$$\frac{}{SKIP \vdash tr = \langle\rangle \vee tr = \langle\checkmark\rangle}$$

Process RUN is able to engage in any trace. For it to satisfy a specification, the specification must allow all possible traces. RUN will therefore satisfy an extremely weak specification, since it will have to places no restrictions on the traces that are acceptable. Such a specification can only be equivalent to $true$:

$$\frac{}{RUN \vdash true(tr)}.$$

2.6 Previous Work

The formal verification field has been enriched by the recent introduction of the class of multi-stack visibly pushdown languages (MVPL). MVPLs are a natural extension of visibly

pushdown languages (VPL) [19]. Both VPL and MVPL have useful applications in the modelling of multithreaded recursive systems, although it could be argued that MVPL models recursive systems in a more expressive and natural manner than the VPL [12, 19]. The MVPL class was first introduced in [19] and is defined using a multi-stack visibly pushdown automaton whose computation is split into k stages such that only one stack can be popped in each stage. Intermediate models also exist, such as the 2-stack visibly pushdown automata (2-VPDA) [12].

Earlier studies [12] showed the 2-VPDA are closed under all Boolean operations and are determinizable in EXPTime. However it was later shown in [18] that 2-vPDAs are undeterminizable and it was instead explained [20] that only the class of languages accepted by multi-stack pushdown automata with a bounded phase are determinizable. In this context a multi-stack PDA (MPDA) is defined similarly to a MVPDA except that the input alphabets no longer determine the stack operations [20]. A bounded phase is a restriction on the computation of class of strings that are accepted by an automaton. For a bounded phase MPDA, a uniform bound k is fixed and only strings that can reduced into k substrings, where each substring will have at most one kind of return node are considered to be in the language accepted by the bounded phase MPDA [20].

Various researchers have also attempted to use VPLs to model and to check equivalence of recursive and concurrent systems. One study [26] examines the equivalence checking on visibly pushdown automata and showed that the complexity (upper and lower bound) for simulation, completed simulation, ready simulation, 2-nested simulation preorders/equivalences and bisimulation equivalence are EXP-time complete. Other research [27] attempted to use VPL to develop a process algebra that would be a superset of CSP, suitable for modelling recursive and concurrent systems. However, as a result of lack of closure under shuffle [11], the resulting process algebra proved to be awkward and of limited use.

Further studies [11] carried out on both VPL and MVPL showed that both languages

were unsuitable for the compositional specification of recursive and concurrent systems because they both lack closure under shuffle. Furthermore, MVPL operations are expressed under strict restrictions on their input alphabets, and removing these restriction will lead to loss of all the interesting closure properties [11, 19].

In the real world concurrent systems execute functions that run in parallel threads of execution that begin identically but behave differently [11]. The stack behaviour of theses parallel threads never overlap, for they normally operate on their own stack [11]. The disjoint operations over MVPL were introduced to model this functionality [11]. The application of disjoint operations on MVPL brings back all the useful closure properties that are lost by removing the rigid restrictions on the original MVPL construction, except closure under shuffle [11]. In addition, [11] introduces a natural renaming process which preserves the closure properties gained from using disjoint operations on MVPLs and also adds closure under shuffle. The renaming process operates by renaming the stack alphabets of a given MVPL, with the restrictions that if a symbol in a stack is renamed, the whole stack is renamed and no symbol associate with the new stack will be in the MVPL after renaming [11].

Finally, part of this dissertation is motivated by research done in both CARET and NWTL temporal logics [1, 2]. Both temporal logics have been proposed for the formal verification of application software as they are able to effectively specify and verify non-regular properties like partial and total correctness and access control properties of application software [1, 2]. All of these efforts pave the way for a fully compositional MVPL-based specification for the verification of recursive and concurrent systems.

Chapter 3

Communicating Multi-Stack Visibly pushdown Processes

A communicating multi-stack visibly pushdown (or CMVP) process is an agent which interacts with its environment (itself also regarded as a process) by performing certain events drawn from a multi-stack visibly pushdown n-stack of call, return alphabets and local symbols : $\Sigma_c = \bigcup_{i=1}^{n} \Sigma_c^i$, $\Sigma_r = \bigcup_{i=1}^{n} \Sigma_r^i$, $\Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_l$. The semantical approach of CMVP will model CSP, so the underlying semantics of CMVP consists in labelled transition systems where states represent CMVP processes. The syntax of CMVP will be based on the following description:

$$S \quad ::= \quad x : A \to S(x) \,|\, S \,\Box\, R \,|\, S \sqcap R \,|\, X \,|\, S \,{}_A\|_B R \,|\, S \setminus A \,|\, \overline{S} \,|\, f(S) \,|\, f^{-1}(S) \,|\, S; R \,|\, S \triangle R$$

where S, R and X range over CMVP processes, x over $\widetilde{\Sigma}$, A and B over $2^{\widetilde{\Sigma}}$, f over the set $\{f : \widetilde{\Sigma} \to \widetilde{\Sigma} : \forall a \in \widetilde{\Sigma}\text{: } f(a), f^{-1}(a) \in \Sigma_c\ [\Sigma_l,\ \Sigma_r]$ iff $a \in \Sigma_c\ [\Sigma_l,\ \Sigma_r] \wedge f^{-1}(a)$ is finite $\wedge\ f(a) = \checkmark$ iff $a = \checkmark \wedge f(a) = \perp$ iff $a = \perp\}$ of $\widetilde{\Sigma}$-transformations. All the common operators between CMVP and CSP have a similar constructions in LTS semantics with the exception of the parallel composition operator. Indeed, the CMVP parallel composition operator applies the stack renaming as described Section 2.4 to the processes it operates on, though this operation is implicit; details are provided in Section 3.1.5. The new operator "Abstract" $\overline{\cdot}$ is also introduced; it can be used to hide the sub-modules of a module (further

discussed in Section 3.1.7). The notion of a module is defined naturally using call and return symbols: A new module is launched once a call c is performed; the execution of that module lasts until the return matching c. The module may perform local actions but also (possibly recursive) calls to other modules (sometimes called "sub-modules" of that module).

The process definitions for both $STOP$ and $SKIP$ are the same in CSP and CMVP. Just like CSP process $STOP$ in CMVP is never prepared to perform any event hence it has no transitions, while $SKIP$ is a state in a process that represents the successful termination of that process (the only event it can perform is the termination event $\checkmark$). Every other CMVP process P_Γ is defined as consisting of an MVPDA state P and a finite number of stacks represented as an n-tuple $\Gamma = (\gamma_1, \ldots, \gamma_n)$. A CMVP process will also be called an LTS state, as opposed to its component P which is a MVPDA state. $P_\epsilon = P_{(\bot,\bot,\ldots,\bot)}$ is defined as a CMVP process P with all its stacks empty. Other than P_ϵ, an LTS state corresponding to a CMVP process will be represented syntactically as $P_{\Gamma(i/\gamma)}$, where P is the current MVPDA state, i represents the stack currently being operated on, $1 \leq i \leq n$, and γ is the current stack prefix of the i-th MVPDA stack. The implicit assumption is that all the other stacks will be unchanged, whereas the prefix γ of the i-th stack will be altered as a result of the current operation (though not even the i-th stack will change otherwise). It should be noted that in our algebra each individual action can only affect a single stack. Therefore our CMVP syntax is refined as follows:

$$\begin{aligned} P_{i/\gamma} \quad ::= \quad & x : A \to P(x)_{i/\gamma} \mid P_{i/\gamma} \,\Box\, Q_{i/\delta} \mid P \sqcap Q \mid N_{i/\gamma} \mid P_{i/\gamma A} \|_B Q_{i/\delta} \mid P \setminus A \\ & \mid \overline{P_{i/\gamma}} \mid f(P_{i/\gamma}) \mid f^{-1}(P_{i/\gamma}) \mid P_{i/\gamma}; Q_{i/\delta} \mid P_{i/\gamma} \triangle Q_{i/\delta} \end{aligned}$$

where P, Q, N range over MVPDA states, i represents the stack being affected by the current operation in the set of stacks Γ and γ and δ represent some (necessarily finite) prefix of the current stack content of the stack in operation i. There is no mention of the stack in operation and stack content in some of the CMVP operations simply because their operations over MVPDA states do not affect the set of stacks. Also, in settling the form

of γ and δ it should be noted that the transitions of a MVPDA (and the associated LTS) depends at most on the top of the stack currently in operation. Therefore, syntactically only the top of the stack in operation is mentioned before and after a call or return transition, while before and after a local transition there is no mention of any of the stacks (since no stack is modified). We thus reach the final syntax of CMVP:

$$\begin{aligned} P' \quad ::= \quad & x : A \to P(x)' \mid P' \,\Box\, Q' \mid P \sqcap Q \mid N' \mid {P'}_A\|_B Q' \mid P \setminus A \\ & \mid \overline{P'} \mid f(P') \mid f^{-1}(P') \mid P'; Q' \mid P' \triangle Q' \end{aligned}$$

We have that $P' ::= P \mid P_{(i/a)}$ and $Q' ::= Q \mid Q_{(i/b)}$, with a and b ranging over $\Gamma \cup \{\varepsilon\}$. When an operator executes a local action we do not mention any stack since we do not operate on any stack. However, if we have have an operation that executes a call or return action we denote the top of the stack by the putting the subscript i/a next to the process, where as stated earlier i would represent the current stack in operation among the set of stacks Γ, and a represents the symbol at the top of the current stack content of the stack in operation i. After the execution of a call or return action there will always be a change (push or pop) in the stack content γ with the exception of when a return action is performed on an empty stack (which will have the symbol $\perp$ at the top of its stack to indicate it is empty).

3.1 The Operational Semantics of CMVP

The subscripts l, c and r are used to represent the sets of local, call and return events, respectively. Hence, any interface $A \in 2^{\widetilde{\Sigma}}$ for n stacks will be the union of $2n + 1$ sets A_l, $(A_c)_i$, $(A_r)_i$, $1 \leq i \leq n$. It should be noted that unlike call and return symbols, local symbols are global to a CMVP process. Hence any call action b that is a member of an interface A is denoted as $b \in (A_c)_i$ meaning that b belongs to the call symbols in interface A and it operates on the i-th stack. Similarly a return action b in an interface A is denoted $b \in (A_r)_i$. A CMVP operation is allowed between CMVP processes only if their stack

alphabets $\widetilde{\Sigma}'$ and $\widetilde{\Sigma}''$ do not overlap[1] (else the main restriction of our MVPL is violated [11]). We will guarantee that the stacks of CMVP processes will not overlap by applying stack renaming to the stacks of one process in the composition of two processes as needed.

The *matched* calls and returns are established by the specification (which will determine which return can pop (or *match*) which call: The matching calls of a return event are defined at specification time by specifying which stack symbols can be popped by the given return. *Balanced* calls and returns on the other hand are determined at run-time: A return *balances* a call if it is labelled as a matching return of that call in the specification and also happens to match that call at run-time. For example, if it is specified that $\{a, b\} \subseteq \Sigma_c^i$ and $c \in \Sigma_r^i$ which will pop either a or b in stack i, then c is the matching return of both a and b; however, during one particular execution is possible that c will only balance a but never b. It must be noted that unbalanced returns are accepted in CMVP processes whenever they appear when the respective stack is empty. In the event that there is an unbalanced return, the (empty) stack content of the CMVP process will remain unchanged.

For ease of presentation we assume without loss of generality that there is a one-to-one mapping between the set of stack symbols and the set of calls, and so a call event for the ith stack will be written a^i and will push a on the i-th stack of that process. Similarly, a return event for stack i will be denoted by b^i. The one-to-one mapping of calls and stack symbols is without loss of generality because we can still specify what can and cannot be popped by a return symbol; the aforementioned one-to-one mapping will thus not limit the capability of defining matched calls and returns. Superscripts are not used on local events because local events are considered to be global and there is no stack manipulation during their execution.

Since CMVP process are able to execute call and return events, CMVP is able to naturally model recursive modules. A module is executed or called by a call event and a return event will signal the return from a module. A local event is used to model all the other

[1]Meaning $\Sigma' \cap \Sigma'' = \emptyset$ for any $\Sigma' \in \{\Sigma_c^i, \Sigma_r^i, 1 \leq i \leq n\}$ and $\Sigma'' \in \{\Sigma_c^k, \Sigma_r^k, 1 \leq k \leq m\}$.

$$\frac{}{(x : A \to P(x))_\Gamma \xrightarrow{a} P(a)_\Gamma} \left[a \in (A_l)\right]$$

$$\frac{}{(x : A \to P(x))_{\Gamma(i/\gamma)} \xrightarrow{a^i} P(a)_{\Gamma(i/a\gamma)}} \left[a^i \in (A_c)_i\right]$$

$$\frac{}{(x : A \to P(x))_{\Gamma(i/a\gamma)} \xrightarrow{a^i} P(a)_{\Gamma(i/\gamma)}} \left[a^i \in (A_r)_i\right]$$

$$\frac{}{(x : A \to P(x))_{\Gamma(i/\perp)} \xrightarrow{a^i} P(a)_{\Gamma(i/\perp)}} \left[a^i \in (A_r)_i\right]$$

Figure 3.1: Prefix choice

actions in the recursive module. This is general, meaning that a call and its corresponding return can happen within another module that is, after another call event happened but before the execution of its balanced return event; this represents a possibly recursive call of a sub-module. In CMVP one call event cannot be used to call two different modules but more than one call event can call the same module.

For the reminder of this paper the stack in operation of CMVP processes will grow to the left, therefore the rightmost place on the stack in operation is reserved for the bottom-of-stack content denoted as $\perp$. As mentioned earlier, given a tuple $\Gamma = (\gamma_1, \gamma_2, \ldots, \gamma_i, \ldots, \gamma_n)$ we denote the tuple $(\gamma_1, \gamma_2, \ldots, \gamma_i, \ldots, \gamma_n)$ by $\Gamma(i/\gamma)$. That is, $\Gamma(i/\gamma)$ is the tuple Γ with the i-th component replaced by γ.

3.1.1 Prefix Choice

The semantics of prefix choice is given in Figure 3.1. Six kinds of syntactic rules introduce the event prefix: $P = a \to P'$, $P = b^i \to P'_{(i/b)}$, $P_{(i/c)} = c^i \to P'_{(i/\perp)}$, $P_{(i/\perp)} = d^i \to P'_{(i/\perp)}$, $P_{(i/\perp)} = STOP$ and $P_{(i/\perp)} = SKIP$. From these syntactic rules, it can be determined that a is local, b^i is a call, c^i is a balanced return, and d^i is an unbalanced return. $P_{(i/\perp)} = STOP$ $[P_{(i/\perp)} = SKIP]$ requires that the process enter the $STOP$ $[SKIP]$ state when the MVPDA state is P and the top of the stack in operation i is empty ($\perp$).

Generally, a system can be specified with as well as without an explicit partitioning of

$$(a)\quad \frac{}{P_\Gamma \xrightarrow{\tau} Q_\Gamma} \qquad (b)\quad \frac{}{P_\Gamma \sqcap Q_\Gamma \xrightarrow{\tau} P_\Gamma} \quad \frac{}{P_\Gamma \sqcap Q_\Gamma \xrightarrow{\tau} Q_\Gamma}$$

Figure 3.2: Internal action (a), internal choice (b)

its events. However, if an explicit partition is not given, then a process can be represented as a sequence of actions (similarly with CSP) only when all the actions are local actions. On the other hand, any finite process (including processes with calls and returns) can be represented as a sequence (desirable in a large system), provided that we specify a partition on its events. For instance, let P_ϵ be the following process without an explicit partition: $P = a \rightarrow P_{(1/a)}$, $P = b \rightarrow P1$, $P1 = e \rightarrow P2_{(1/e)}$, $P2 = d \rightarrow P3$, $P3_{(1/e)} = f \rightarrow P4_{(1/\perp)}$, $P4_{(1/a)} = c \rightarrow P4_{(1/\perp)}$, $P4_{(1/\perp)} = STOP$. The process can be written with an explicit partition $A_l = \{b, d\}$, $(A_c)_1 = \{a^1, e^1\}$, $(A_r)_1 = \{c^1, f^1\}$ as follows: $P_{(1/\perp)} = a^1 \rightarrow P_{(1/a)}$, $P = b \rightarrow e^1 \rightarrow d \rightarrow P3_{(1/ea)}$, $P3_{(1/ea)} = f^1 \rightarrow P4_{(1/a)}$, $P4_{(1/a)} = c^1 \rightarrow P4_{(1/\perp)}$, $P4_{(1/\perp)} = STOP$.

3.1.2 Internal Event

A CMVP process can perform actions not noticeable to the environment. These actions are called internal and are denoted by the symbol τ. Internal actions can change the current MVPDA state of a process but will not change the stack content of any stack. The behaviour of the τ transition is described in Figure 3.2(a).

3.1.3 Choice

The semantics of internal and external choice are given in Figures 3.2(b) and 3.3, respectively. The choice operator does not modify the matched calls and returns. The set of stacks of the composite process in a choice construct is also similar to the set of stacks of the component processes of the construct. A process that chooses (once!) between '$]$' and '$\rangle$' as balanced return for '$[$' can be defined as follows: $P = [^1 \rightarrow\ P_{(1/[)}$, $P_{(1/[)} =]^1 \rightarrow\ P1_{(1/\perp)}$, $P_{(1/[)} = \rangle^1 \rightarrow\ P2_{(1/\perp)}$, $P1_{(1/\perp)} = STOP, P2_{(1/\perp)} = STOP$. Note

$$\frac{P_\Gamma \xrightarrow{\tau} P'_\Gamma}{\begin{array}{l} P_\Gamma \,\square\, Q_\Gamma \xrightarrow{\tau} P'_\Gamma \,\square\, Q_\Gamma \\ Q_\Gamma \,\square\, P_\Gamma \xrightarrow{\tau} Q_\Gamma \,\square\, P'_\Gamma \end{array}} \qquad \frac{P_\Gamma \xrightarrow{a} P'_\Gamma}{\begin{array}{l} P_\Gamma \,\square\, Q_\Gamma \xrightarrow{a} P'_\Gamma \\ Q_\Gamma \,\square\, P_\Gamma \xrightarrow{a} P'_\Gamma \end{array}} [a \in (A_l)]$$

$$\frac{P_{\Gamma(i/\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/a\gamma)}}{\begin{array}{l} P_{\Gamma(i/\gamma)} \,\square\, Q_\Gamma \xrightarrow{a^i} P'_{\Gamma(i/a\gamma)} \\ Q_\Gamma \,\square\, P_{\Gamma(i/\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/a\gamma)} \end{array}} [a^i \in (A_c)_i] \qquad \frac{P_{\Gamma(i/a\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/\gamma)}}{\begin{array}{l} P_{\Gamma(i/a\gamma)} \,\square\, Q_\Gamma \xrightarrow{a^i} P'_{\Gamma(i/\gamma)} \\ Q_\Gamma \,\square\, P_{\Gamma(i/\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/\gamma)} \end{array}} [a^i \in (A_r)_i]$$

$$\frac{P_{\Gamma(i/\perp)} \xrightarrow{a^i} P'_{\Gamma(i/\perp)}}{\begin{array}{l} P_{\Gamma(i/\perp)} \,\square\, Q_{\Gamma(i/\perp)} \xrightarrow{a^i} P'_{\Gamma(i/\perp)} \\ Q_{\Gamma(i/\perp)} \,\square\, P_{\Gamma(i/\perp)} \xrightarrow{a^i} P'_{\Gamma(i/\perp)} \end{array}} [a^i \in (A_r)_i] \qquad \frac{P_\Gamma \xrightarrow{\checkmark} P'_\Gamma}{\begin{array}{l} P_\Gamma \,\square\, Q_\Gamma \xrightarrow{\checkmark} P'_\Gamma \\ Q_\Gamma \,\square\, P_\Gamma \xrightarrow{\checkmark} P'_\Gamma \end{array}}$$

Figure 3.3: External choice

$$\frac{P_\Gamma \xrightarrow{\tau} P'_\Gamma}{N_\Gamma \xrightarrow{\tau} P'_\Gamma} \; \big[N = P\big] \qquad \frac{P_\Gamma \xrightarrow{a} P'_\Gamma}{N_\Gamma \xrightarrow{a} P'_\Gamma} \; \big[a \in (A_l), N = P\big]$$

$$\frac{P_{\Gamma(i/\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/a\gamma)}}{N_{\Gamma(i/\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/a\gamma)}} \; \big[a^i \in (A_c)_i, N = P\big] \qquad \frac{P_{\Gamma(i/a\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/\gamma)}}{N_{\Gamma(i/a\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/\gamma)}} \; \big[a^i \in (A_r)_i, N = P\big]$$

$$\frac{P_{\Gamma(i/\perp)} \xrightarrow{a^i} P'_{\Gamma(i/\perp)}}{N_{\Gamma(i/\perp)} \xrightarrow{a^i} P'_{\Gamma(i/\perp)}} \; \big[a^i \in (A_r)_i, N = P\big] \qquad \frac{P_\Gamma \xrightarrow{\checkmark} P'_\Gamma}{N_\Gamma \xrightarrow{\checkmark} P'_\Gamma} \; \big[N = P\big]$$

Figure 3.4: Recursion

that ‘]’ and ‘⟩’ are both matching returns of ‘[’ (since they both pop it from the stack), but only one is used as a balanced return, depending on the environment. In contrast, a process Q defined with the following rules: $Q = [^1 \rightarrow \; Q_{(1/[)}$, $Q = \tau \rightarrow \; Q1$, $Q = \tau \rightarrow \; Q2$, $Q1_{(1/[)} =]^1 \rightarrow Q1_{(1/\perp)}, Q2_{(1/[)} =\rangle^1 \rightarrow \; Q2_{(1/\perp)}, Q1_{(1/\perp)} = STOP, Q2_{(1/\perp)} = STOP$, makes an internal choice to transition from a state Q to either states $Q1$ or $Q2$ independent of the environment, and so the decision of whether ‘]’ or ‘⟩’ matches ‘[’ does not depend on the environment anymore.

3.1.4 Recursion

Recursion in CSP and CMVP are defined similarly, however, a CSP recursive process creates loops among LTS states while CMVP recursive processes creates loops among MVPDA states and can perform recursive function calls using call events. A recursive MVPDA state may perform a sequence of calls and returns in addition to local actions before transitioning back into itself, so that each recursive loop can change the stack content of the stack in operation. If the changes made to the stack content of the set of stacks is zero (i.e. if the executed events are all local events), then the recursive process can be represented by a finite state machine just like in CSP. The condition for a CMVP process P_ϵ to be called recursive is that the process definition will contain the MVPDA state P performing a sequence of events and transitioning to the same state P.

The semantics of recursion is shown in Figure 3.4. Consider as an example the following CMVP recursive processes, which produces balanced parentheses: $P = ({}^1\!\!\rightarrow P_{(1/()}, P_{(1/()} =)^1 \rightarrow P_{(1/\perp)}, P_{(1/\perp)} = STOP$. P_ϵ can produce an infinite number of LTS states and infinitely many possible traces although we only have one MVPDA state P. A recursive process has an iteratively increasing part followed by an iteratively decreasing part. The increasing part may run at infinitum and can produce an unbounded stack. A stack inspection interrupt process as defined in Section 3.1.9 can be used to manage recursive processes and transfer the execution of control from one process to another.

Here is how a CMVP process P_ϵ can produce the trace (()()):

$$\begin{aligned} P_\epsilon &= ({}^1\!\!\rightarrow P_{((\perp)} = ({}^1\!\!\rightarrow ({}^1\!\!\rightarrow P_{(((\perp)} = ({}^1\!\!\rightarrow ({}^1\!\!\rightarrow)^1 \rightarrow P_{((\perp)} = ({}^1\!\!\rightarrow ({}^1\!\!\rightarrow)^1 \rightarrow ({}^1\!\!\rightarrow P_{(((\perp)} \\ &= ({}^1\!\!\rightarrow ({}^1\!\!\rightarrow)^1 \rightarrow ({}^1\!\!\rightarrow)^1 \rightarrow P_{((\perp)} = ({}^1\!\!\rightarrow ({}^1\!\!\rightarrow)^1 \rightarrow ({}^1\!\!\rightarrow)^1 \rightarrow)^1 \rightarrow P_{(\perp)} \\ &= ({}^1\!\!\rightarrow ({}^1\!\!\rightarrow)^1 \rightarrow ({}^1\!\!\rightarrow)^1 \rightarrow)^1 \rightarrow STOP \end{aligned}$$

Consider now the process $Q = ({}^1\!\!\rightarrow Q1_{(1/()}, Q1_{(1/()} =)^1 \rightarrow Q_{(1/\perp)}, Q_{(i/\perp)} = STOP$. It may have infinitely long traces and can be written as follows: $Q_\epsilon = ({}^1\!\!\rightarrow)^1 \rightarrow Q, Q_{(1/\perp)} = STOP$. An argument can be made that the events "$({}^1$" and "$)^1$" are behaving like locals in Q_ϵ, hence,

$$\frac{P_\Gamma \xrightarrow{a} P'_\Gamma \qquad Q_{\Gamma'} \xrightarrow{a} Q'_{\Gamma'}}{(P_A\|_B Q)_{\Gamma\cdot\Gamma'} \xrightarrow{a} (P'_A\|_B Q')_{\Gamma\cdot\Gamma'}} \ \left[a \in A_l \cap B_l\right]$$

$$\frac{P_\Gamma \xrightarrow{a} P'_\Gamma}{\begin{array}{l}(P_A\|_B Q)_{\Gamma\cdot\Gamma'} \xrightarrow{a} (P'_A\|_B Q)_{\Gamma\cdot\Gamma'} \\ (Q_B\|_A P)_{\Gamma'\cdot\Gamma} \xrightarrow{a} (Q_B\|_A P')_{\Gamma'\cdot\Gamma}\end{array}} \ \left[a \in A_l \setminus B_l\right]$$

$$\frac{P_{\Gamma(i/\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/a\gamma)}}{\begin{array}{l}(P_A\|_B Q)_{(\Gamma\cdot\Gamma')(i/\gamma)} \xrightarrow{a^i} (P'_A\|_B Q)_{(\Gamma\cdot\Gamma')(i/a\gamma)} \\ (Q_B\|_A P)_{(\Gamma'\cdot\Gamma)(n+i/\gamma)} \xrightarrow{a^{n+i}} (Q_B\|_A P')_{(\Gamma'\cdot\Gamma)(n+i/a\gamma)}\end{array}} \ \left[\begin{array}{l}a^i \in (A_c)_i \setminus B \\ \implies a^{n+i} \in (A_c)_{n+i} \setminus B\end{array}\right]$$

$$\frac{P_{\Gamma(i/a\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/\gamma)}}{\begin{array}{l}(P_A\|_B Q)_{(\Gamma\cdot\Gamma')(i/a\gamma)} \xrightarrow{a^i} (P'_A\|_B Q)_{(\Gamma\cdot\Gamma')(i/\gamma)} \\ (Q_B\|_A P)_{(\Gamma'\cdot\Gamma)(n+i/a\gamma)} \xrightarrow{a^{n+i}} (Q_B\|_A P')_{(\Gamma'\cdot\Gamma)(n+i/\gamma)}\end{array}} \ \left[\begin{array}{l}a^i \in (A_r)_i \setminus B \\ \implies a^{n+i} \in (A_r)_{n+i} \setminus B\end{array}\right]$$

$$\frac{P_{\Gamma(i/\bot)} \xrightarrow{a^i} P'_{\Gamma(i/\bot)}}{\begin{array}{l}(P_A\|_B Q)_{(\Gamma\cdot\Gamma')(i/\bot)} \xrightarrow{a^i} (P'_A\|_B Q_{(\Gamma\cdot\Gamma')(i/\bot)} \\ (Q_B\|_A P)_{(\Gamma'\cdot\Gamma)(n+i/\bot)} \xrightarrow{a^{n+i}} (Q_B\|_A P')_{(\Gamma'\cdot\Gamma)(n+i/\bot)}\end{array}} \ \left[\begin{array}{l}a^i \in (A_r)_i \setminus B \\ \implies a^{n+i} \in (A_r)_{n+i} \setminus B\end{array}\right]$$

$$\frac{P_\Gamma \xrightarrow{\tau} P'_\Gamma}{\begin{array}{l}(P_A\|_B Q)_{\Gamma\cdot\Gamma'} \xrightarrow{\tau} (P'_A\|_B Q)_{\Gamma\cdot\Gamma'} \\ (Q_B\|_A P)_{\Gamma'\cdot\Gamma} \xrightarrow{\tau} (Q_B\|_A P')_{\Gamma'\cdot\Gamma}\end{array}}$$

$$\frac{P_\Gamma \xrightarrow{\checkmark} P'_\Gamma \qquad Q_{\Gamma'} \xrightarrow{\checkmark} Q'_{\Gamma'}}{(P_A\|_B Q)_{\Gamma\cdot\Gamma'} \xrightarrow{\checkmark} (P'_A\|_B Q')_{\Gamma\cdot\Gamma'}}$$

Figure 3.5: Alphabetized parallel

Q_ϵ can be represented by a finite state machine. However, in P_ϵ above the event "(" *must* be a call and the event ")" *must* be a return.

3.1.5 Parallel Composition

Figure 3.5 shows the semantics of parallel composition. Since having an overlap in call and return stack alphabets of MVPDAs violates the restriction of our process algebra, CMVP processes in a parallel composition can only synchronize over local events. To guarantee that there will be no stack overlap between the processes being operated on, CMVP parallel composition is implemented using disjoint operations [11]. To use these disjoint operations

over CMVP on two processes in a parallel composition construct, a stack renaming must be applied to one of the two processes. The stack renaming action performed by the CMVP parallel composition operator allows our process algebra to have the required closure properties needed to define a parallel composition between CMVP processes. The CMVP parallel composition is denoted ${}_A\|_B$, with A and B being the respective event interfaces of the processes on either side of the parallel composition construct. The notation $\Gamma \cdot \Gamma'$ represent the concatenation of the two tuples Γ and Γ'.

Now, recall that every CMVP process has a finite number of stacks Γ which will be represented as an n-tuple $(\gamma_1, \ldots, \gamma_n)$. If there is a parallel composition between two processes P_Γ and $Q_{\Gamma'}$ (where Γ and Γ' represents the set of stacks of P and Q respectively), the set of stacks of the second process in the parallel composition must be renamed. Hence we rename the stacks of process $Q_{\Gamma'}$ using the concept of *stack renaming* [11]:

We have that the $1, \ldots, n$-*Stack Renaming* $\mathcal{R}_{1,\ldots,n}(Q)$ of $Q_{\Gamma'}$ is an MVPDA over the n-stack call-return alphabet $\widetilde{\Sigma'_{n+x}} = \{(\Sigma_c^{n+i}, \Sigma_r^{n+i})_{(n+1)\leq(n+i)\leq(n+x)}, \Sigma_l\}$ where $n = n$ in $P_{\Gamma(1,\ldots,n)}$, such that there exists a bijection $f : \Sigma_c^i \cup \Sigma_r^i \longrightarrow \Sigma_c^{n+i} \cup \Sigma_r^{n+i}$ where $(n+1) \leq (n+i) \leq (n+x)$, with $f(x) \in \Sigma_c^{n+i}$ iff $x \in \Sigma_c^i$ and $f(x) \in \Sigma_r^{n+i}$ iff $x \in \Sigma_r^i$.

With the renaming of the stacks of $Q_{\Gamma'}$, the sequence of the stacks become $\Gamma' = ((n+1), \ldots, (n+x))$ while the sequence of the stacks of process P_Γ remain $\Gamma = (1, \ldots, n)$. After all the stacks of $Q_{\Gamma'}$ have been successfully renamed, the parallel composition between both processes can be executed and the resulting set of stacks will be the concatenation $(\Gamma \cdot \Gamma')$ of both set of stacks of process P_Γ and $Q_{\Gamma'}$. We are guaranteed that there will be no stack overlap and we also have the added advantage of having an organised set of stacks which will aid us in monitoring the activities of call and return symbols.

In a parallel composition any common local event of the component processes synchronizes during execution. However such a synchronized execution does not change the content of any stack. The component process perform independently any events (call, return and unsynchronized local events) which are not common to the parallel composition. Thus every

$$\frac{P_\Gamma \xrightarrow{t} P'_\Gamma}{P_\Gamma \setminus B \xrightarrow{t} P'_\Gamma \setminus B} \left[t \in \{\tau, \checkmark\}\right] \qquad \frac{P_\Gamma \xrightarrow{a} P'_\Gamma}{P_\Gamma \setminus B \xrightarrow{\tau} P'_\Gamma \setminus B} \left[a \in A_l \cap B\right]$$

$$\frac{P_\Gamma \xrightarrow{a} P'_\Gamma}{P_\Gamma \setminus B \xrightarrow{a} P'_\Gamma \setminus B} \left[a \in A_l \setminus B\right] \qquad \frac{P_{\Gamma(i/\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/a\gamma)}}{P_{\Gamma(i/\gamma)} \setminus B \xrightarrow{a^i} P'_{\Gamma(i/a\gamma)} \setminus B} \left[a^i \in (A_c)_i\right]$$

$$\frac{P_{\Gamma(i/a\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/\gamma)}}{P_{\Gamma(i/a\gamma)} \setminus B \xrightarrow{a^i} P'_{\Gamma(i/\gamma)} \setminus B} \left[a^i \in (A_r)_i\right] \qquad \frac{P_{\Gamma(i/\perp)} \xrightarrow{a^i} P'_{\Gamma(i/\perp)}}{P_{\Gamma(i/\perp)} \setminus B \xrightarrow{a^i} P'_{\Gamma(i/\perp)} \setminus B} \left[a^i \in (A_r)_i\right]$$

Figure 3.6: Hiding

interleaved call and return event pushes or pops one stack symbol to or from the stack in operation of the process executing the unsynchronized event.

3.1.6 Hiding

The semantics for hiding is given in Figure 3.6. Hiding does not change the stack content of the set of stacks of a process because only local symbols can be hidden. If a hidden local event is executed the process is seen to perform an internal action τ (i.e. the local action becomes invisible to the environment), and then a change in state occurs. Since this operation does not affect the stacks of the process, hiding cannot change the balance of a process or the matched calls and returns in the stacks of a process.

3.1.7 Abstract

The abstract operator hides all the sub-modules of a module. The motivation for this operator is the abstract path in CARET and NWTL [1, 2]. By executing the abstract operator sub-modules can be hidden from the environment; it allows for the abstraction of call and return symbols. This cannot be achieved using the hide operator since only local symbols can be hidden. The local trace of a module can be defined using the abstract operator, allowing for the specification of the internal properties of recursive modules. The semantics of abstract is given in Figure 3.7. When a call symbol is executed, abstract

$$\frac{P_{\Gamma(i/b\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/ab\gamma)}}{\begin{array}{l}\overline{P_{\Gamma(i/b\gamma)}} \xrightarrow{a^i} \overline{P'_{\Gamma(i/\bar{a}b\gamma)}} \\ \overline{P_{\Gamma(i/\tilde{b}\gamma)}} \xrightarrow{\tau} \overline{P'_{\Gamma(i/\bar{a}\tilde{b}\gamma)}} \\ \overline{P_{\Gamma(i/\bar{b}\gamma)}} \xrightarrow{\tau} \overline{P'_{\Gamma(i/\bar{a}\bar{b}\gamma)}}\end{array}} [a^i \in (A_c)_i] \qquad \frac{P_{\Gamma(i/a\gamma)} \xrightarrow{b} P'_{\Gamma(i/a\gamma)}}{\begin{array}{l}\overline{P_{\Gamma(i/a\gamma)}} \xrightarrow{b} \overline{P'_{\Gamma(i/a\gamma)}} \\ \overline{P_{\Gamma(i/\tilde{a}\gamma)}} \xrightarrow{\tau} \overline{P'_{\Gamma(i/\tilde{a}\gamma)}} \\ \overline{P_{\Gamma(i/\bar{a}\gamma)}} \xrightarrow{\tau} \overline{P'_{\Gamma(i/\bar{a}\gamma)}}\end{array}} [b \in (A_l)]$$

$$\frac{P_{\Gamma(i/a\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/\gamma)}}{\begin{array}{l}\overline{P_{\Gamma(i/a\gamma)}} \xrightarrow{a^i} \overline{P'_{\Gamma(i/\gamma)}} \\ \overline{P_{\Gamma(i/\tilde{a}\gamma)}} \xrightarrow{a^i} \overline{P'_{\Gamma(i/\gamma)}} \\ \overline{P_{\Gamma(i/\bar{a}\gamma)}} \xrightarrow{\tau} \overline{P'_{\Gamma(i/\gamma)}}\end{array}} \left[a^i \in (A_r)_i\right] \qquad \frac{P_{\Gamma(i/\perp)} \xrightarrow{a^i} P'_{\Gamma(i/\perp)}}{\overline{P_{\Gamma(i/\perp)}} \xrightarrow{a^i} P'_{\Gamma(i/\perp)}} \left[a^i \in (A_r)_i\right]$$

$$\frac{P_\Gamma \xrightarrow{\tau} P'_\Gamma}{\overline{P_\Gamma} \xrightarrow{\tau} \overline{P'_\Gamma}} \qquad \frac{P_\Gamma \xrightarrow{\checkmark} P'_\Gamma}{\overline{P_\Gamma} \xrightarrow{\checkmark} P'_\Gamma}$$

$$\frac{P_{\Gamma(k/\gamma)} \xrightarrow{a^k} P'_{\Gamma(k/a\gamma)}}{\begin{array}{l}\overline{P_{\Gamma(i/\tilde{b}\gamma)}} \xrightarrow{a^k} \overline{P'_{\Gamma(k/a\gamma)}} \\ \overline{P_{\Gamma(i/\bar{a}\tilde{b}\gamma)}} \xrightarrow{a^k} \overline{P'_{\Gamma(k/a\gamma)}} \\ \overline{P_{\Gamma(i/\bar{b}\gamma)}} \xrightarrow{a^k} \overline{P'_{\Gamma(k/a\gamma)}}\end{array}} [a^k \in (A_c)_k] \qquad \frac{P_{\Gamma(k/a\gamma)} \xrightarrow{a^k} P'_{\Gamma(k/\gamma)}}{\begin{array}{l}\overline{P_{\Gamma(i/\tilde{b}\gamma)}} \xrightarrow{a^k} \overline{P'_{\Gamma(k/\gamma)}} \\ \overline{P_{\Gamma(i/\bar{a}\tilde{b}\gamma)}} \xrightarrow{a^k} \overline{P'_{\Gamma(k/\gamma)}} \\ \overline{P_{\Gamma(i/\bar{b}\gamma)}} \xrightarrow{a^k} \overline{P'_{\Gamma(k/\gamma)}}\end{array}} [a^k \in (A_r)_k]$$

Figure 3.7: Abstract

pushes the corresponding stack symbol to the stack in operation with two special markers: $\tilde{\cdot}$ denotes the internal call of the main module and $\bar{\cdot}$ denotes the internal call of a sub-module. If the top of the stack in operation contains any special marker then every local event will be hidden; calls and returns are pushed to/popped off the stack in operation but are otherwise hidden as well (except for top-level calls and returns in the module). If a return symbol is executed, and the top of the stack in operation is not marked, then the process goes out of abstraction.

Let B (with call b^i and return f^i), and C (with call d^i and return e^i) be two modules. The top-level process P calls B and B calls C: $P = a \rightarrow Q$, $Q = b^i \rightarrow R_{(i/b)}$, $R = c \rightarrow S$, $S = d^i \rightarrow T_{(i/d)}$, $T = c \rightarrow U$, $U_{(i/d)} = e^i \rightarrow V_{(i/\perp)}$, $V_{(i/b)} = f^i \rightarrow W_{(i/\perp)}$, and $W = c \rightarrow STOP$. The sub-modules of B can be hidden by using abstract: $\overline{P_{(\gamma_1,\ldots,\gamma_n)}} =$

$$\frac{P_{\Gamma(i/\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/a\gamma)}}{f(P)_{\Gamma(i/\gamma)} \xrightarrow{f(a)^i} f(P')_{\Gamma(i/f(a)\gamma)}} [a^i \in (A_c)_i] \qquad \frac{P_\Gamma \xrightarrow{\tau} P'_\Gamma}{f(P)_\Gamma \xrightarrow{\tau} f(P')_\Gamma}$$

$$\frac{P_{\Gamma(i/a\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/\gamma)}}{f(P)_{\Gamma(i/f(a)\gamma)} \xrightarrow{f(a)^i} f(P')_{\Gamma(i/\gamma)}} [a^i \in (A_r)_i] \qquad \frac{P_\Gamma \xrightarrow{a} P'_\Gamma}{f(P)_\Gamma \xrightarrow{f(a)} f(P')_\Gamma} [a \in (A_l)]$$

$$\frac{P_{\Gamma(i/\perp)} \xrightarrow{a^i} P'_{\Gamma(i/\perp)}}{f(P)_{\Gamma(i/\perp)} \xrightarrow{f(a)^i} f(P')_{\Gamma(i/\perp)}} [a^i \in (A_r)_i] \qquad \frac{P_\Gamma \xrightarrow{\checkmark} P'_\Gamma}{f(P)_\Gamma \xrightarrow{\checkmark} f(P')_\Gamma}$$

Figure 3.8: Forward renaming

$a \to \overline{Q_{(\gamma_1,\ldots,\gamma_n)}} \to b^i \to \overline{R_{(\gamma_1,\ldots,\gamma_{i-1},\tilde{b}\perp,\gamma_{i+1},\ldots,\gamma_n)}} \to c \to \overline{S_{(\gamma_1,\ldots,\gamma_{i-1},\tilde{b}\perp,\gamma_{i+1},\ldots,\gamma_n)}} = d^i \to \overline{T_{(\gamma_1,\ldots,\gamma_{i-1},\overline{d\tilde{b}}\perp,\gamma_{i+1},\ldots,\gamma_n)}} \to c \to \overline{U_{(\gamma_1,\ldots,\gamma_{i-1},\overline{d\tilde{b}}\perp,\gamma_{i+1},\ldots,\gamma_n)}} \to e^i \to \overline{V_{(\gamma_1,\ldots,\gamma_{i-1},\tilde{b}\perp,\gamma_{i+1},\ldots,\gamma_n)}} \to f^i \to W_{(\gamma_1,\ldots,\gamma_{i-1},\perp,\gamma_{i+1},\ldots,\gamma_n)} \to c \to STOP$. We actually hide sub-module C in this particular example. A run of P without the abstract operator will be as follows: $P_{(\gamma_1,\ldots,\gamma_n)} = a \to Q_{(\gamma_1,\ldots,\gamma_n)} \to b^i \to R_{(\gamma_1,\ldots,\gamma_{i-1},b\perp,\gamma_{i+1},\ldots,\gamma_n)} \to c \to S_{(\gamma_1,\ldots,\gamma_{i-1},b\perp,\gamma_{i+1},\ldots,\gamma_n)} \to d^i \to T_{(\gamma_1,\ldots,\gamma_{i-1},db\perp,\gamma_{i+1},\ldots,\gamma_n)} \to c \to U_{(\gamma_1,\ldots,\gamma_{i-1},db\perp,\gamma_{i+1},\ldots,\gamma_n)} \to e^i \to V_{(\gamma_1,\ldots,\gamma_{i-1},b\perp,\gamma_{i+1},\ldots\gamma_n)} \to f^i \to W_{(\gamma_1,\ldots,\gamma_{i-1},\perp,\gamma_{i+1},\ldots,\gamma_{i+1}n)} \to c \to STOP$.

3.1.8 Renaming

The semantics of forward and backward renaming are given in Figures 3.8 and 3.9, respectively. Renaming can change the matched calls and returns of a CMVP process, however it cannot modify the MVPL partition. For instance, if we have a call event a^i with a matching return event b^i, if a^i is renamed $f(a)^i$, then b^i will be the matching return event for $f(a)^i$. There might be no "reverse" renaming that retrieves the original process or set of matched call-returns: Suppose that a renaming f is applied to the return event $\rangle$ in process P_ϵ in our previous example illustrating choice (in Section 3.1.3) such that $f(\rangle) =]$. We then get a process whose traces define the language $[^n]^n$; no renaming can give back the original.

$$\frac{P_{\Gamma(i/\gamma)} \xrightarrow{f(a)^i} P'_{\Gamma(i/f(a)\gamma)}}{f^{-1}(P)_{\Gamma(i/\gamma)} \xrightarrow{a^i} f^{-1}(P')_{\Gamma(i/a\gamma)}} [a^i \in (A_c)_i] \qquad \frac{P_\Gamma \xrightarrow{\tau} P'_\Gamma}{f^{-1}(P)_\Gamma \xrightarrow{\tau} f^{-1}(P')_\Gamma}$$

$$\frac{P_{\Gamma(i/f(a)\gamma)} \xrightarrow{f(a)^i} P'_{\Gamma(i/\gamma)}}{f^{-1}(P)_{\Gamma(i/a\gamma)} \xrightarrow{a^i} f^{-1}(P')_{\Gamma(i/\gamma)}} [a^i \in (A_r)_i] \qquad \frac{P_\Gamma \xrightarrow{f(a)} P'_\Gamma}{f^{-1}(P)_\Gamma \xrightarrow{a} f^{-1}(P')_\Gamma} [a \in (A_l)]$$

$$\frac{P_{\Gamma(i/\perp)} \xrightarrow{f(a)^i} P'_{\Gamma(i/\perp)}}{f^{-1}(P)_{\Gamma(i/\perp)} \xrightarrow{a^i} f^{-1}(P')_{\Gamma(i/\perp)}} [a^i \in (A_r)_i] \qquad \frac{P_\Gamma \xrightarrow{\checkmark} P'_\Gamma}{f^{-1}(P)_\Gamma \xrightarrow{\checkmark} f^{-1}(P')_\Gamma}$$

Figure 3.9: Backward renaming

$$\frac{P_{\Gamma(i/\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/a\gamma)}}{P_{\Gamma(i/\gamma)}; Q_{\Gamma'} \xrightarrow{a^i} P'_{\Gamma(i/a\gamma)}; Q_{\Gamma'}} [a^i \in (A_c)_i] \qquad \frac{P_\Gamma \xrightarrow{\tau} P'_\Gamma}{P_\Gamma; Q_{\Gamma'} \xrightarrow{\tau} P'_\Gamma; Q_{\Gamma'}}$$

$$\frac{P_{\Gamma(i/a\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/\gamma)}}{P_{\Gamma(i/a\gamma)}; Q_{\Gamma'} \xrightarrow{a^i} P'_{\Gamma(i/\gamma)}; Q_{\Gamma'}} [a^i \in (A_r)_i] \qquad \frac{P_\Gamma \xrightarrow{a} P'_\Gamma}{P_\Gamma; Q_{\Gamma'} \xrightarrow{a} P'_\Gamma; Q_{\Gamma'}} [a \in (A_l)]$$

$$\frac{P_{\Gamma(i/\perp)} \xrightarrow{a^i} P'_{\Gamma(i/\perp)}}{P_{\Gamma(i/\perp)}; Q_{\Gamma'} \xrightarrow{a^i} P'_{\Gamma(i/\perp)}; Q_{\Gamma'}} [a^i \in (A_r)_i] \qquad \frac{P_\Gamma \xrightarrow{\checkmark} P'_\Gamma}{P_\Gamma; Q_{\Gamma'} \xrightarrow{\tau} Q_{\Gamma'}}$$

Figure 3.10: Sequential composition

3.1.9 Sequential Composition and Interrupt

A CMVP process can continue to execute indefinitely, retaining control over execution throughout. By applying sequential composition or interrupt, the control of execution can be passed from one CMVP process to a second CMVP process, either because the first process reaches a particular point in its execution (i.e. termination) where it is prepared to release control (in the case of sequential composition), or because the second process demands it (in the case of interrupt).

A sequential composition $P_\Gamma;Q_{\Gamma'}$ allows the first process P_Γ to execute till the point of termination and then it gives the the control of execution to the second process $Q_{\Gamma'}$ in the sequential composition construct. When the first process terminates, its termination event

$$\frac{P_\Gamma \xrightarrow{\tau} P'_\Gamma}{P_\Gamma \triangle Q_{\Gamma'} \xrightarrow{\tau} P'_\Gamma \triangle Q_{\Gamma'}} \qquad \frac{P_\Gamma \xrightarrow{a} P'_\Gamma}{P_\Gamma \triangle Q_{\Gamma'} \xrightarrow{a} P'_\Gamma \triangle Q_{\Gamma'}}[a \in (A_l)]$$

$$\frac{P_{\Gamma(i/\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/a\gamma)}}{P_{\Gamma(i/\gamma)} \triangle Q_{\Gamma'} \xrightarrow{a^i} P'_{\Gamma(i/a\gamma)} \triangle Q_{\Gamma'}}[a \in (A_c)_i] \qquad \frac{P_{\Gamma(i/a\gamma)} \xrightarrow{a^i} P'_{\Gamma(i/\gamma)}}{P_{\Gamma(i/a\gamma)} \triangle Q_{\Gamma'} \xrightarrow{a^i} P'_{\Gamma(i/\gamma)} \triangle Q_{\Gamma'}}[a \in (A_r)_i]$$

$$\frac{P_{\Gamma(i/\perp)} \xrightarrow{a^i} P'_{\Gamma(i/\perp)}}{P_{\Gamma(i/\perp)} \triangle Q_{\Gamma'} \xrightarrow{a^i} P'_{\Gamma(i/\perp)} \triangle Q_{\Gamma'}}[a \in (A_r)_i] \qquad \frac{P_\Gamma \xrightarrow{\checkmark} P'_\Gamma}{P_\Gamma \triangle Q_{\Gamma'} \xrightarrow{\checkmark} P'_\Gamma}$$

$$\frac{Q_{\Gamma'} \xrightarrow{\tau} Q'_{\Gamma'}}{P_\Gamma \triangle Q_{\Gamma'} \xrightarrow{\tau} Q'_{\Gamma'}} \qquad \frac{Q_{\Gamma'(j/\gamma)} \xrightarrow{a^j} Q'_{\Gamma'(j/a\gamma)}}{P_\Gamma \triangle Q_{\Gamma'(j/\gamma)} \xrightarrow{a^j} Q'_{\Gamma'(j/a\gamma)}}[a \in (A_c)_j]$$

$$\frac{Q_{\Gamma'} \xrightarrow{a} Q'_{\Gamma'}}{P_\Gamma \triangle Q_{\Gamma'} \xrightarrow{a} Q'_{\Gamma'}}[a \in (A_l)] \qquad \frac{Q_{\Gamma'(j/a\gamma)} \xrightarrow{a^j} Q'_{\Gamma'(j/\gamma)}}{P_\Gamma \triangle Q_{\Gamma'(j/a\gamma)} \xrightarrow{a^j} Q'_{\Gamma'(j/\gamma)}}[a \in (A_r)_j]$$

$$\frac{Q_{\Gamma'(j/\perp)} \xrightarrow{a^j} Q'_{\Gamma'(j/\perp)}}{P_\Gamma \triangle Q_{\Gamma'(j/\perp)} \xrightarrow{a^j} Q'_{\Gamma'(j/\perp)}}[a \in (A_r)_j]$$

Figure 3.11: Interrupt

becomes internal to the composition, since $P_\Gamma;Q_{\Gamma'}$ should not indicate that it has finished until $Q_{\Gamma'}$ finally terminates.

On the other hand, an interrupt $P_\Gamma \triangle Q_{\Gamma'}$, allows $Q_{\Gamma'}$ to take control from P_Γ at any point in its execution. The interrupting process $Q_{\Gamma'}$ may begin execution at any point throughout the execution of P_Γ : the performance of $Q_{\Gamma'}$'s first event is the point at which control passes, and the first process P_Γ is discarded. Figures 3.10 and 3.11 show the semantics of sequential composition and interrupt, respectively.

Unlike parallel composition, neither the sequential composition nor the interrupt operators use the stack renaming feature explicitly. We argue that the set of stacks of processes in a sequential composition and interrupt construct will never overlap, since in both operators once control of execution is passed onto the second process the first process can no longer perform any actions. Hence once the control of execution is passed on to the second process

of the constructs the set of stacks of the first process in the construct do not change (and therefore can be discarded).

3.2 CMVP Is a Process Algebra

Theorem 3.2.1 *CMVP is an algebra; that is, CMVP is closed under all its operators. The underlying semantics of any CMVP process is a MVPDA (or an equivalent LTS).*

Proof. We proceed by structural induction. *STOP*, *SKIP* are defined as CMVP processes similarly to their definition in CSP. Also, CMVP's closure under prefix choice, external choice, internal choice, recursion, renaming, hiding and abstract are all immediate. Prefix choice follows the definition of a transition in the associated MVPDA. External choice is simply a prefix choice with more than one alternative (i.e. this allows more than one transition out of one state). The internal choice construct is connection between two LTS to a common start state via τ transitions (this does not change the stack contents of the set of stacks). Recursion is simply a loop from some MVPDA state P back into the same state P. The set of stacks of the MVPDA are manipulated according to the MVPDA semantics introduced by the other transitions and this does not introduce infinite MVPDA states. Renaming cannot changed the MVPL partition. Closure under hiding is straightforward since only local symbols can be hidden; τ transitions replace hidden local symbols while a change in state occurs in the LTS. When an abstract operation is applied on an MVPDA, the MVPDA transitions are hidden (that is, replaced with τ) after the first stack symbol is pushed into the stack in operation. The MVPDA comes out of the abstracted state after the first stack symbol which was pushed to the stack in operation is popped. Hence, the abstract construct replaces whole portions of the LTS with τ transitions.

Let now P and Q be two CMVP processes whose semantics are given by the MVPDAs L' and L'', respectively. Let L' and L'' have initial states I' and I'' and final states H' and H'', respectively.

Consider now the sequential composition of P and Q. An MVPDA L corresponding to this sequential composition can be constructed as follows: In a sequential composition L' will run to completion, followed by a run of L'' (which is only possible when L' has reached termination, case in which there is no further computation relying on the set of stacks of L'). We then take the disjoint union of the stacks of L' and L'' and call the results the set of stacks of L. The transitions of L will then be the union of the transition relations of L' and L'' (the latter suitably modified by the disjoint union operation), plus ε-transitions from all the states in H' to I''. The initial state of L is I' and the final states are exactly all the states from H''. That the resulting MVPDA is the semantic model of the sequential composition is immediate: The automaton L' runs until it reaches its final state. Once this happens, the control is given to the state I'', in effect launching L''. The disjoint union ensures that the stacks of L' are never used from this point on, and that the stacks of L'' are empty when the control is given to this automaton (since no transitions from L' operates on them), as desired.

Closure under interrupt is shown by a similar construction, with the exception that the control can be passed from P to Q or equivalently from L' to L'' at any time, hence in addition to the union of the transitions of L' and L'' (as for the sequential composition) we add one ε-transition from every state of L' to I'' (instead of ε-transition from the states in H' to I'' only). As above, once the control passes to L'' the stacks of L' are not longer needed, and at that moment the stacks of L'' are all empty, as desired.

The construction of the parallel composition L of L' and L'' is given as follows: the Cartesian product of the MVPDA states in L' and L'' is the set of MVPDA states in L, with the initial state of L' and the final states of L'' being the initial and final states of L, respectively. The set of stacks of L will be the disjoint union of the set of stacks of L' and L''. Recall that CMVP parallel composition makes use of stack renaming [11] which implements such a disjoint union. Hence we are guaranteed that there will be no stack overlap between L' and L'' and so the restriction on CMVP processes is not violated. The

transition relation of L is defined as follows:

- For synchronized local symbols l we have the transition $(P', \varepsilon) \xrightarrow{l} (Q', \varepsilon)$ in L' and the transition $(P'', \varepsilon) \xrightarrow{l} (Q'', \varepsilon)$ in L'' by the inductive hypothesis. In this case we add to L the transition $((P', P''), \varepsilon) \xrightarrow{l} ((Q', Q''), \varepsilon)$. For unsynchronized local symbols that have the transition $(P', \varepsilon) \xrightarrow{l} (Q', \varepsilon)$ in L' [the transition $(P'', \varepsilon) \xrightarrow{l} (Q'', \varepsilon)$ in L''] we add to L the transitions $((P', X), \varepsilon) \xrightarrow{l} ((Q', X), \varepsilon)$ for all the states X of L'' [$((X, P''), \varepsilon) \xrightarrow{l} ((X, Q''), \varepsilon)$ for all the states X of L']. The correctness of this construction is immediate from the fact that there is no change in the content of any stack for local symbols, while the transitions above ensures that the two components of the state of L change according to the semantics of L' and L'', respectively.

- All the calls c^i of L' and c^{n+i} of L'' (after the stack renaming) and return symbols r^i in L' and r^{n+i} in L'' (again after the stack renaming) are unsynchronized. These are integrated in L as follows:

  For every set of rules $(P', \varepsilon) \xrightarrow{c^i} (Q', a)$, $(P'', \varepsilon) \xrightarrow{c^{n+i}} (Q'', b)$, $(R', a) \xrightarrow{r^i} (S', \varepsilon)$, and $(R'', b) \xrightarrow{r^{n+i}} (S'', \varepsilon)$ we add the following rules, respectively: $((P', X), \varepsilon) \xrightarrow{c^i} ((Q', X), a)$, $((Y, P''), \varepsilon) \xrightarrow{c^{n+i}} ((Y, Q''), b)$, $((R', X), a) \xrightarrow{r^i} ((S', X), \varepsilon)$, and $((Y, R''), b) \xrightarrow{r^{n+i}} ((Y, S''), \varepsilon)$ for all states X of L'' and Y of L'. Once more the correctness of the construction follows from the fact that the sets of stacks of L' and L'' do not overlap now that the stacks of L'' have been renamed.

- No other transitions are included in the transition relation of L. ■

Chapter 4

A Detailed Example

In this section we illustrate the use of CMVP in the specification and analysis of a concurrent and recursive system. MVPDAs are used to model the concurrent and recursive systems. We explain how each CMVP operations can be used to describe and analyze concurrent and recursive systems.

Let P and Q be two recursive systems that run concurrently (i.e. on independent sets of stacks). We represent P and Q as CMVP rocessess with the same name (P_ϵ and Q_ϵ). Both P_ϵ and Q_ϵ have two stacks each named 1 and 2. They also have the same local symbols (i.e. a, c, e), call symbols (i.e. b^1, f^2) and return symbols (d^1, g^2). In both CMVP proceses, the call b^1 pushes b to stack 1 and the return d^1 pops b, while the call f^2 pushes f to stack 2 and the return g^1 pops f.

The differences between both processes are their states and their transition relations. Process P_ϵ has the states P, $P2$, $P3$, $P4$, $P5$, $P6$, $P7$ and $P8$. The following is the CMVP definition of the process: $(P = a \rightarrow P2)$, $(P2 = c \rightarrow P4)$, $(P2_{1/\perp} = b^1 \rightarrow P3_{1/f})$, $(P3 = e \rightarrow P)$, $(P4_{2/\perp} = f^2 \rightarrow P_{2/f})$, $(P4_{2/f} = g^2 \rightarrow P6_{2/\perp})$, $(P4_{2/\perp} = g^2 \rightarrow P6_{2/\perp})$, $(P2_{1/b} = d^1 \rightarrow P5_{1/\perp})$, $(P2_{1/\perp} = d^1 \rightarrow P5_{1/\perp})$, $(P5_{1/b} = d^1 \rightarrow P5_{1/\perp})$, $(P5_{1/\perp} = d^1 \rightarrow P5_{1/\perp})$, $(P5 = c \rightarrow P6)$, $(P5_{1/b} = d^1 \rightarrow P7_{1/\perp})$, $(P5_{1/\perp} = d^1 \rightarrow P7_{1/\perp})$, $(P6_{2/f} = g^2 \rightarrow P6_{2/\perp})$, $(P6_{2/\perp} = g^2 \rightarrow P6_{2/\perp})$, $(P6_{2/f} = g^2 \rightarrow P8_{2/\perp})$, $(P6_{2/\perp} = g^2 \rightarrow P8_{2/\perp})$, $(P7 = a \rightarrow P8)$.

On the other hand, the process Q_ϵ features the states Q, $Q2$, $Q3$, $Q4$, $Q5$, $Q6$, $Q7$ and $Q8$.

The following is the definition of Q_ϵ: $(Q = a \to Q2)$, $(Q2 = c \to Q3)$, $(Q3_{1/\perp} = b^1 \to Q_{1/b})$, $(Q3_{1/b} = d^1 \to Q5_{1/\perp})$, $(Q3_{1/\perp} = d^1 \to Q5_{1/\perp})$, $(Q3 = e \to Q4)$, $(Q4_{2/\perp} = f^2 \to Q2_{2/f})$, $(Q4_{2/f} = g^2 \to Q6_{2/\perp})$, $(Q4_{2/\perp} = g^2 \to Q6_{2/\perp})$, $(Q5_{1/b} = d^1 \to Q5_{1/\perp})$, $(Q5_{1/\perp} = d^1 \to Q5_{1/\perp})$, $(Q5 = c \to Q6)$, $(Q5_{1/b} = d^1 \to Q7_{1/\perp})$, $(Q5_{1/\perp} = d^1 \to Q7_{1/\perp})$, $(Q6_{2/f} = g^2 \to Q6_{2/\perp})$, $(Q6_{2/\perp} = g^2 \to Q6_{2/\perp})$, $(Q6_{2/f} = g^2 \to Q8_{2/\perp})$, $(Q6_{2/\perp} = g^2 \to Q8_{2/\perp})$, $(Q7 = a \to Q8)$.

4.1 Prefix Choice

All states except $P8$ and $Q8$ in the process P and Q respectively are defined using the prefix choice operator. Take state $P5$ in Figure 4.1 as an example; we have that state $P5$ can execute a return action d^1 that returns the symbol b if it is at the top of the stack 1 of P, else it performs an empty return. It can then transition to the state $P5$ or transition to the state $P7$. Also state $P5$ can execute a local action c and transitions to a state $P6$.

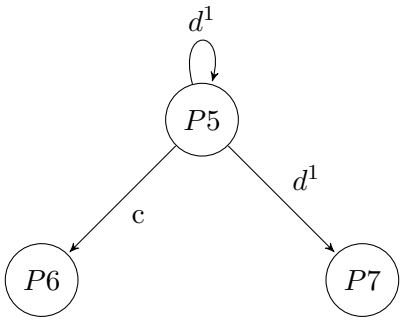

Figure 4.1: Illustration of the possible transitions of state $P5$ of process P_ϵ

4.2 Internal Event

An internal event is an action that is executed by a system but that is not noticeable to the environment. In CMVP internal actions can change the state of a process in a system but they cannot affect the stack content of the system. Since CMVP can specify processes in a compositional manner, systems can be broken down into smaller components. Let us

assume now that both process P_ϵ and Q_ϵ are components of a larger process H. If process H makes an internal choice between state P of P_ϵ and state Q of Q_ϵ as illustrated in Figure 4.2, then $H = \tau \to P$ and $H = \tau \to Q$.

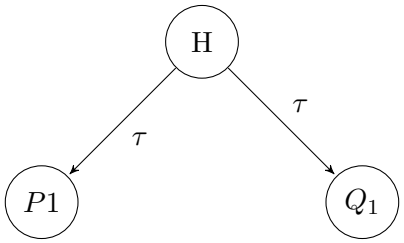

Figure 4.2: Illustration of internal choice: $P \sqcap Q$

4.3 Choice

There are numerous examples of states using the external choice operator in both process P_ϵ and Q_ϵ based on their transition rules. Recall that the external choice operator allows states to make transitions between two or more states depending on the execution of a external event (i.e local, call or return action).

Figure 4.1 illustrates that state $P5$ can make an external choice between three states: state $P5$ if it performs the return event d^1 by using the transition rules ($P5_{1/\perp} = d^1 \to P5_{1/\perp}$) or ($P5_{1/b} = d^1 \to P5_{1/\perp}$), state $P6$ if it performs the local event c by using the transition rule ($P5 = c \to P6$) and state $P7$ if it performs the return event d^1 by using the transition rules ($P5_{1/\perp} = d^1 \to P7_{1/\perp}$) or ($P5_{1/b} = d^1 \to P7_{1/\perp}$). Therefore, we have that state $P5$ makes an external choice to transition in either states $P5$, $P6$, or $P7$ depending on the event being performed.

In Figure 4.2, state H makes an internal choice between events two states P and Q by executing an internal event τ.

4.4 Recursion

An instance of recursion can be seen in Figure 4.1. State $P5$ of process P can perform the return event d^1 and remain in the same state by using the transition rules ($P5_{1/\perp} = d^1 \rightarrow P5_{1/\perp}$) or ($P5_{1/b} = d^1 \rightarrow P5_{1/\perp}$).

Other examples of recursive states in P_ϵ and Q_ϵ based on their transition rules are $P6$, $Q5$ and $Q6$. All theses states perform actions that allow them to transition back into themselves. Recursive states can also execute call events (which pushes symbols to the stack) and return events (which pops symbols from the stack).

4.5 Parallel Composition

A CMVP parallel composition between P_ϵ and Q_ϵ, $P_\epsilon \| Q_\epsilon$ will have the CMVP operator rename the set of stacks of Q (the second process in the parallel composition construct). Both processes have two stacks each named 1 and 2. The sets of stacks of Q_ϵ are renamed as follows: the $1,2$-*Stack Renaming* $\mathcal{R}_{1,2}(Q)$ of Q_Γ is an MVPDA over the 2-stack call-return alphabet $\widetilde{\Sigma'_{2+x}} = \{(\Sigma_c^{2+i}, \Sigma_r^{2+i})_{(2+1)\leq(2+i)\leq(2+x)}, \Sigma_l\}$ such that there exists a bijection $f : \Sigma_c^i \cup \Sigma_r^i \longrightarrow \Sigma_c^{2+i} \cup \Sigma_r^{2+i}$ where $(2+1) \leq (2+i) \leq (2+x)$, with $f(x) \in \Sigma_c^{2+i}$ iff $x \in \Sigma_c^i$ and $f(x) \in \Sigma_r^{2+i}$ iff $x \in \Sigma_r^i$. Therefore we have stack 1 of Q_Γ becoming stack $2+1 = 3$ and stack 2 of Q_Γ becoming stack $2+2 = 4$. The parallel composition operator renamed the set of stacks of process Q and so the transitions of Q are implicitly modified as follows: ($Q = a \rightarrow Q2$), ($Q2 = c \rightarrow Q3$), ($Q3_{3/\perp} = b^3 \rightarrow Q_{3/b}$), ($Q3_{3/b} = d^3 \rightarrow Q5_{3/\perp}$), ($Q3_{3/\perp} = d^3 \rightarrow Q5_{3/\perp}$), ($Q3 = e \rightarrow Q4$), ($Q4_{4/\perp} = f^4 \rightarrow Q2_{4/f}$), ($Q4_{4/f} = g^4 \rightarrow Q6_{4/\perp}$), ($Q4_{4/\perp} = g^4 \rightarrow Q6_{4/\perp}$), ($Q5_{3/b} = d^3 \rightarrow Q5_{3/\perp}$), ($Q5_{3/\perp} = d^3 \rightarrow Q5_{3/\perp}$), ($Q5 = c \rightarrow Q6$), ($Q5_{3/b} = d^3 \rightarrow Q7_{3/\perp}$), ($Q5_{3/\perp} = d^3 \rightarrow Q7_{3/\perp}$), ($Q6_{4/f} = g^4 \rightarrow Q6_{4/\perp}$), ($Q6_{4/\perp} = g^4 \rightarrow Q6_{4/\perp}$), ($Q6_{4/f} = g^4 \rightarrow Q8_{4/\perp}$), ($Q6_{4/\perp} = g^4 \rightarrow Q8_{4/\perp}$), ($Q7 = a \rightarrow Q8$).

Once stack renaming is done, the parallel composition operator concatenates the set of stacks of both processes. Figure 4.3 shows a a run of the parallel composition of P and Q.

$$(P\|Q)_{(\perp,\perp,\perp,\perp)} \xrightarrow{a} (P2\|Q2)_{(\perp,\perp,\perp,\perp)} \xrightarrow{c} (P4\|Q3)_{(\perp,\perp,\perp,\perp)} \xrightarrow{f^2} (P\|Q3)_{(\perp,f\perp,\perp,\perp)} \xrightarrow{b^3}$$
$$(P\|Q)_{(\perp,f\perp,b\perp,\perp)} \xrightarrow{a} (P2\|Q2)_{(\perp,f\perp,b\perp,\perp)} \xrightarrow{b^1} (P3\|Q2)_{(b\perp,f\perp,b\perp,\perp)} \xrightarrow{c}$$
$$(P3\|Q3)_{(b\perp,f\perp,b\perp,\perp)} \xrightarrow{e} (P\|Q4)_{(b\perp,f\perp,b\perp,\perp)} \xrightarrow{f^4} (P\|Q2)_{(b\perp,f\perp,b\perp,f\perp)} \xrightarrow{a}$$
$$(P2\|Q2)_{(b\perp,f\perp,b\perp,f\perp)} \xrightarrow{c} (P4\|Q3)_{(b\perp,f\perp,b\perp,f\perp)} \xrightarrow{f^2} (P\|Q3)_{(b\perp,ff\perp,b\perp,f\perp)} \xrightarrow{b^3}$$
$$(P\|Q)_{(b\perp,ff\perp,bb\perp,f\perp)} \xrightarrow{a} (P2\|Q2)_{(b\perp,ff\perp,bb\perp,f\perp)} \xrightarrow{c} (P4\|Q3)_{(b\perp,ff\perp,bb\perp,f\perp)} \xrightarrow{g^2}$$
$$(P6\|Q3)_{(b\perp,f\perp,bb\perp,f\perp)} \xrightarrow{d^3} (P6\|Q5)_{(b\perp,f\perp,b\perp,f\perp)} \xrightarrow{g^2} (P6\|Q5)_{(b\perp,\perp,b\perp,f\perp)} \xrightarrow{d^3}$$
$$(P6\|Q5)_{(b\perp,\perp,\perp,f\perp)} \xrightarrow{g^2} (P8\|Q5)_{(b\perp,\perp,\perp,f\perp)} \xrightarrow{d^3} (P8\|Q7)_{(b\perp,\perp,\perp,f\perp)}$$

Figure 4.3: A run of the parallel composition between processes P_ϵ and Q_ϵ

As shown in Figure 4.3, both P and Q synchronize on the local symbols a, c and e. However, there is no synchronization when P or Q execute call or return actions. It can also be noticed in Figure 4.3 that P and Q do not execute some local actions at the same time. This only occurs because of the current states that both processes are at during the execution of the local actions. There is an interleaving execution of the local actions only because one of the processes cannot execute the local action that is being offered to the parallel composition construct at its current state. As far as both processes P and Q are in a states that can perform the local action being offered, synchronization will occur.

4.6 Hiding

Recall that only local symbols can be hidden using the hiding operator in CMVP. Hence the hiding operator does not affect the stack content of a process. When the hiding operator is applied on a process, the hidden action behaves like an internal action τ thus becoming invisible to the environment. Figure 4.4 illustrates the possible initial transitions of $P5\backslash\{c\}$.

4.7 Abstract

The run in Figure 4.5 demonstrates the use of the abstract operator in a run of P in comparison to an un-abstracted run. By using the abstract operator local, call and return

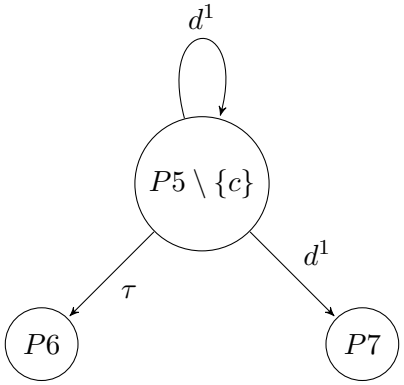

Figure 4.4: Illustration of the possible transitions of state $P5 \setminus \{c\}$

symbols can hidden (or abstracted). As shown in Figure 4.5 a process will get out of abstraction only when a return action is executed and the stack symbol with the $\tilde{\cdot}$ symbol has been popped from the stack.

$$\overline{P1}_{(\bot,\bot)} \xrightarrow{a} \overline{P2}_{(\bot,\bot)} \xrightarrow{b^1} \overline{P3}_{(\tilde{b}\bot,\bot)} \xrightarrow{\tau} \overline{P1}_{(\tilde{b}\bot,\bot)} \xrightarrow{\tau} \overline{P2}_{(\tilde{b}\bot,\bot)} \xrightarrow{\tau} \overline{P3}_{(\tilde{b}\tilde{b}\bot,\bot)} \xrightarrow{\tau}$$
$$\overline{P1}_{(\tilde{b}\tilde{b}\bot,\bot)} \xrightarrow{\tau} \overline{P2}_{(\tilde{b}\tilde{b}\bot,\bot)} \xrightarrow{\tau} \overline{P5}_{(\tilde{b}\bot,\bot)} \xrightarrow{d^1} \overline{P5}_{(\bot,\bot)} \xrightarrow{d^1} P7_{(\bot,\bot)}$$

(a) Abstracted configuration

$$P1_{(\bot,\bot)} \xrightarrow{a} P2_{(\bot,\bot)} \xrightarrow{b^1} P3_{(b\bot,\bot)} \xrightarrow{e} P1_{(b\bot,\bot)} \xrightarrow{a} P2_{(b\bot,\bot)} \xrightarrow{b^1}$$
$$P3_{(bb\bot,\bot)} \xrightarrow{e} P1_{(bb\bot,\bot)} \xrightarrow{a} P2_{(bb\bot,\bot)} \xrightarrow{d^1} P5_{(b\bot,\bot)} \xrightarrow{d^1} P5_{(\bot,\bot)} \xrightarrow{d^1} P7_{(\bot,\bot)}$$

(b) Un-abstracted configuration

Figure 4.5: An abstracted configuration and an un-abstracted configuration of P_ϵ

4.8 Renaming

Recall that the renaming operator is able to change the matched call-return when it is applied. If renaming is applied to the call action b^1 in process P it will changed the matched call-return (recall that b^1 pushes b to stack 1, while d^1 pops b off stack 1), leading to a modification of the transitions of P. After the renaming operator is applied to the call

action b^1 in process P, $f(b^1)$ pushes $f(b)$ to stack 1 while d^1 pops $f(b)$ off stack 1.

4.9 Sequential Composition and Interrupt

A run of the sequential composition between P and Q is illustrated in Figure 4.6. As soon as P finishes executing the control of execution is passed on to Q. That is, when P reaches its final state control is passed to Q through an internal action τ as illustrated in the figure. $P;Q$ does not signal termination until Q reaches termination.

$$P_{(\perp,\perp)};Q_{(\perp,\perp)} \xrightarrow{a} P2_{(\perp,\perp)};Q_{(\perp,\perp)} \xrightarrow{b^1} P3_{(b\perp,\perp)};Q_{(\perp,\perp)} \xrightarrow{e} P_{(b\perp,\perp)};Q_{(\perp,\perp)} \xrightarrow{a}$$
$$P2_{(b\perp,\perp)};Q_{(\perp,\perp)} \xrightarrow{d^1} P5_{(\perp,\perp)};Q_{(\perp,\perp)} \xrightarrow{d^1} P7_{(\perp,\perp)};Q_{(\perp,\perp)} \xrightarrow{\tau} P7_{(\perp,\perp)};Q_{(\perp,\perp)} \xrightarrow{a}$$
$$P7_{(\perp,\perp)};Q2_{(\perp,\perp)} \xrightarrow{c} P7_{(\perp,\perp)};Q3_{(\perp,\perp)} \xrightarrow{b^3} P7_{(\perp,\perp)};Q_{(b\perp,\perp)} \xrightarrow{a} P7_{(\perp,\perp)};Q2_{(b\perp,\perp)} \xrightarrow{c}$$
$$P7_{(\perp,\perp)};Q3_{(b\perp,\perp)} \xrightarrow{d^3} P7_{(\perp,\perp)};Q5_{(\perp,\perp)} \xrightarrow{d^3} P7_{(\perp,\perp)};Q7_{(\perp,\perp)}$$

Figure 4.6: A run illustrating a sequential composition between P_ϵ and Q_ϵ

The interrupt operator on the other hand allows control of execution to be passed from one process to another at an arbitrary point of an execution. Figure 4.7 illustrates a possible run of an interrupt between P and Q.

$$P_{(\perp,\perp)} \triangle Q_{(\perp,\perp)} \xrightarrow{a} P2_{(\perp,\perp)} \triangle Q_{(\perp,\perp)} \xrightarrow{b^1} P3_{(b\perp,\perp)} \triangle Q_{(\perp,\perp)} \xrightarrow{e} P_{(b\perp,\perp)} \triangle Q_{(\perp,\perp)} \xrightarrow{a}$$
$$P2_{(b\perp,\perp)} \triangle Q_{(\perp,\perp)} \xrightarrow{a} P2_{(b\perp,\perp)} \triangle Q2_{(\perp,\perp)} \xrightarrow{c} P2_{(b\perp,\perp)} \triangle Q3_{(\perp,\perp)} \xrightarrow{b^1} P2_{(b\perp,\perp)} \triangle Q_{(b\perp,\perp)} \xrightarrow{a}$$
$$P2_{(b\perp,\perp)} \triangle Q2_{(b\perp,\perp)} \xrightarrow{c} P2_{(b\perp,\perp)} \triangle Q3_{(b\perp,\perp)} \xrightarrow{d^1} P2_{(b\perp,\perp)} \triangle Q5_{(\perp,\perp)} \xrightarrow{d^1} P2_{(b\perp,\perp)} \triangle Q7_{(\perp,\perp)}$$

Figure 4.7: A run illustrating an interrupt between P_ϵ and Q_ϵ

Process P does not need to reach a final state before it passes control to process Q. As illustrated in the figure, control is passed from process P to process Q when Q interrupts P at state $P2$ by executing the event a. As soon as Q executes the event a, no further computation is done on the set of stacks of P. The interrupt operator allows control of execution to be passes from one process to the other at any point in its execution.

Chapter 5

CMVP Trace Semantics

CMVP trace semantics generates the set of traces of a CMVP process: if P_Γ is CMVP process then traces(P_Γ) is the set of traces it can produce. The process definitions for *STOP* and *SKIP*, are the same in CSP and CMVP. *RUN* however cannot be defined as a CMVP process that can be combined with other processes (and thus be useful in establishing laws and other properties), since having an overlap in call and return stack alphabets of MVPDAs violates the restriction of our process algebra. So instead, we define a CMVP process (RUN_{Σ_l}) equivalent to the CSP process *RUN* in CMVP.

RUN_{Σ_l} is defined to be the process with an interface Σ_l that can always perform any local event at any given time. Furthermore RUN_{Σ_l} cannot execute any call or return action and so its stacks will always be empty. We therefore do not mention explicitly its set of stacks. We then have:

$$\text{traces}(RUN_{\Sigma_l}) = \{tr | tr \in TRACE \wedge \sigma(tr) \subseteq \Sigma_l\}$$

5.1 Prefix Choice

There are only two possibilities when observing the process $((x : A \rightarrow P(x))_{\Gamma(i/\gamma)}$: either no event has executed, or an event $a \in A$ has executed, making its subsequent behaviour that of the corresponding process $P(a)_{\Gamma(i/\gamma')}$. If $a \in A_l$ then $\gamma' = \gamma$:

$$\text{traces}((x : A \rightarrow P(x))_\Gamma) = \{\langle\rangle\} \cup \{\langle a \rangle.tr | a \in A_l \wedge tr \in \text{traces}(P(a))_\Gamma\}$$

If $a \in (A_c)_i$ then $a = a^i$ and $\gamma' = a\gamma$:

$$\text{traces}((x : A \to P(x))_{\Gamma(i/\gamma)}) = \{\langle\rangle\} \cup \{\langle a\rangle.tr | a \in A_c \wedge tr \in \text{traces}(P(a)_{\Gamma(i/a\gamma)})\}$$

If $a \in (A_r)_i$ and $\gamma = a\delta$ then $a = a^i$ and $\gamma' = \delta$:

$$\text{traces}((x : A \to P(x))_{\Gamma(i/a\delta)}) = \{\langle\rangle\} \cup \{\langle a\rangle.tr | a \in A_r \wedge tr \in \text{traces}(P(a)_{\Gamma(i/\delta)})\}$$

If $a \in (A_r)_i$ and $\gamma = \perp$ then then $a = a^i$ and $\gamma' = \gamma$:

$$\text{traces}((x : A \to P(x))_{\Gamma(i/\perp)}) = \{\langle\rangle\} \cup \{\langle a\rangle.tr | a \in A_r \wedge tr \in \text{traces}(P(a)_{\Gamma(i/\perp)})\}$$

In considering a general notation for the above rules we note that $(x : A \to P(x))_{\Gamma(i/\gamma)} \xrightarrow{a \in A} P(a)_{\Gamma(i/\gamma')}$, meaning that after the event $x \in A$ is executed the stack in operation will be γ', where γ' will depend on the type of x (call, return, local) in the usual manner described earlier. Hence the above four rules can be written as the single rule as follows:

$$\begin{aligned} \text{traces}((x : A \to P(x))_{\Gamma(i/\gamma)}) \quad = \quad \{\langle\rangle\} \quad \cup \quad \{\langle a\rangle.tr | a \in A \wedge \\ (x : A \to P(x))_{\Gamma(i/\gamma)} \xrightarrow{a \in A} P(a)_{\Gamma(i/\gamma')} \\ \wedge tr \in \text{traces}(P(a)_{\Gamma(i/\gamma')})\} \end{aligned}$$

If $P_{(aa\perp)}$ and $P_{(a\perp)}$ are CMVP processes having the same process definition: where $P_{(1/a)} = a \to P_{(1/\perp)}$, $P_{(1/\perp)} = STOP$. The $\text{traces}(P_{(aa\perp)}) = \{\langle\rangle, \langle a\rangle, \langle a, a\rangle\}$ and $\text{traces}(P_{(a\perp)}) = \{\langle\rangle, \langle a\rangle\}$. It is clear that they are not equivalent processes since $\text{traces}(P_{(aa\perp)}) \neq \text{traces}(P_{(a\perp)})$.

5.2 External Choice

When observing the choice construct $P_\Gamma \,\square\, Q_{\Gamma'}$, there are only two possibilities: either an execution of P_Γ will occur or an execution of $Q_{\Gamma'}$ will occur. The choice operator when applied will split a process into alternative processes, allowing the parent process to choose

once between one of the alternative processes. The alternative processes operate on the same set of stacks as the parent process.

$$\text{traces}(P_\Gamma \,\Box\, Q_{\Gamma'}) = \text{traces}(P_\Gamma) \cup \text{traces}(Q_{\Gamma'})$$

Figure 5.1 enumerates the laws of external choice. The first three laws are derived from the properties of the disjoint union operator. Law $\Box - unit$ states that any process P_Γ will always be given precedence over $STOP$ when there is a choice construct between both processes. In algebraic terms, $STOP$ is a unit of external choice.

$$P_\Gamma \,\Box\, P_\Gamma = P_\Gamma \qquad \Box - idem$$

$$P_\Gamma \,\Box\, (Q_{\Gamma'} \,\Box\, R_{\Gamma''}) = (P_\Gamma \,\Box\, Q_{\Gamma'}) \,\Box\, R_{\Gamma''} \qquad \Box - assoc$$

$$P_\Gamma \,\Box\, Q_{\Gamma'} = Q_{\Gamma'} \,\Box\, P_\Gamma \qquad \Box - sym$$

$$P_\Gamma \,\Box\, STOP = P_\Gamma \qquad \Box - unit$$

Figure 5.1: Laws for external choice

5.3 Internal Choice

The internal choice construct $P_\Gamma \sqcap Q_{\Gamma'}$ will behaves either as P_Γ or as $Q_{\Gamma'}$. It performs a hidden action (τ) thereby preventing its environment from exercising control over its behaviour. The traces of internal choice are as follows:

$$\text{traces}(P_\Gamma \sqcap Q_{\Gamma'}) = \text{traces}(P_\Gamma) \cup \text{traces}(Q_{\Gamma'})$$

Figure 5.2 enumerates the law of internal choice. Unlike the external choice construct $P_\Gamma \,\Box\, Q_{\Gamma'}$, the environment of an internal choice construct plays no part in its outcome. As a result, both constructs have different executions in resolving a choice. However, the major concern of a trace observer is identifying the sequence of possible actions of the choice outcomes. Therefore the Law $choice - equiv$ states that the sequence of possible actions of

an internal choice construct and an external choice construct with the same set of CMVP processes are equal.

$$P_\Gamma \,\Box\, Q_{\Gamma'} = P_\Gamma \sqcap Q_{\Gamma'} \qquad\qquad choice - equiv$$

Figure 5.2: Law for internal choice

5.4 Parallel Composition

A parallel composition $(P_A\|_B Q)_{\Gamma.\Gamma'}$ consists of P_Γ executing events in an interface A, and $Q_{\Gamma'}$ executing events in an interface B (recall that a stack renaming occurs on one of the two processes in a parallel composition). The processes P_Γ and $Q_{\Gamma'}$ synchronize only on events in $A_l \cap B_l$ (i.e. they synchronize on local event common to both A and B), and execute other events in their respective interfaces independently. Any execution of the parallel composition projected onto interface A must be an event that P_Γ can execute. Similarly, any execution projected onto interface B must be an event that $Q_{\Gamma'}$ can execute. Therefore, the traces of $(P_A\|_B Q)_{\Gamma.\Gamma'}$ are those sequences of events that P_Γ and $Q_{\Gamma'}$ can execute which are in interface A and interface B, and also termination. Hence, the set of events in the trace $(\sigma(tr))$, must be contained in $(A \cup B)^\checkmark$.

$$\begin{aligned}\text{traces}(P_A\|_B Q)_{\Gamma.\Gamma'} = \{tr \in TRACE \quad | \quad & tr \upharpoonright A^\checkmark \in \text{traces}(P_\Gamma) \wedge tr \upharpoonright B^\checkmark \in \text{traces}(Q_{\Gamma'}) \\ & \wedge \sigma(tr) \subseteq (A \cup B)^\checkmark\}\end{aligned}$$

Figure 5.3 enumerates the laws of alphabetized parallel. In both Law $\| - step$ and Law $\| - term$ 2, γ and γ' represent the stack content of stack i (and $n+i$) before and after an execution (or transition): if a local or an empty return event is executed then $\gamma' = \gamma$, else if a call or an un-empty return event is executed them $\gamma' \neq \gamma$ (in the execution of a call a symbol is pushed to the stack and with a return a symbol is popped from the stack). Law $\| - assoc$ is a form of association derived from the disjoint union operator. The outcome of the parallel composition construct in Law $\| - assoc$ is independent of the order in which

$$(P_A\|_{B\cup C}(Q_B\|_C R))_{\Gamma.\Gamma'.\Gamma''} = ((P_A\|_B Q)_{A\cup B}\|_C R)_{\Gamma.\Gamma'.\Gamma''} \qquad \|-assoc$$

$$(P_A\|_B Q)_{\Gamma.\Gamma'} = (Q_B\|_A P)_{\Gamma'.\Gamma} \qquad \|-sym$$

$$\begin{aligned}
&C\subseteq A \wedge D\subseteq B \Rightarrow ((x:C\to P(x)_A\|_B(y:D\to Q(y))_{\Gamma.\Gamma'}\\
&= z:((C\setminus B)\cup(D\setminus A)\cup(C\cap D))\to R(z)_{(\Gamma(1,\ldots,n)').(\Gamma'(n+1,\ldots,x)')}\\
&\text{where } R(z)_{(\Gamma(1,\ldots,n)')\cup(\Gamma'(n+1,\ldots,x)')}\\
&\qquad = (P(c)_A\|_B(y:D\to Q(y))_{\Gamma.\Gamma'(i/\gamma')}) \quad \text{if} \quad c\in C\setminus B\\
&\qquad = ((x:C\to P(x))_A\|_B Q(c))_{\Gamma.\Gamma'(n+i/\gamma')} \quad \text{if} \quad c\in D\setminus A\\
&\qquad = (P(c)_A\|_B Q(c))_{\Gamma.\Gamma'} \quad \text{if} \quad c\in C_l\cap D_l \qquad \|-step
\end{aligned}$$

$$SKIP_A\|_B SKIP = SKIP \qquad \|-term\ 1$$

$$\begin{aligned}
&((x:C\to P(x))_A\|_B SKIP)_{\Gamma(i/\gamma)} = ((x:C\to P(x))_A\|_B SKIP)_{\Gamma(i/\gamma')}\\
&= x:C\cap(A\setminus B)\to(P(x)_A\|_B SKIP)_{\Gamma(i/\gamma')} \qquad \|-term\ 2
\end{aligned}$$

$$(P_A\|_A(RUN_{\Sigma_l}))_\Gamma = P_\Gamma \qquad \|-unit$$

Figure 5.3: Laws for alphabetized parallel

components of the parallel composition construct are composed together. Law $\|-unit$ describes a unit for the parallel composition construct: process RUN_{Σ_l} is always ready to execute any local event in the common interface, hence there are no restriction on P_Γ's execution of those events. Law $\|-step$ breaks down parallel composition into prefix choice (i.e. all the possible actions that a parallel composition can execute). It also shows that CMVP processes can only have synchronization over local events, and explains how the set of stacks in the parallel composition construct are manipulated. Laws $\|-term$ 1 and $\|-term$ 2 explain the termination of a parallel composition construct. Termination occurs only when all processes in the parallel composition construct are ready to terminate.

It should be noted that since synchronization only occurs over local events, the parallel composition construct in CMVP does not have a zero (as opposed to the process $STOP$ being a zero for parallel composition in CSP). Indeed, if a process P is in a parallel composition with $STOP$, synchronization over local symbols will occur as before, and given that $STOP$

$$(P_\Gamma \setminus A) \setminus B = P_\Gamma \setminus (A \cup B) \qquad hide - combine$$

$$STOP \setminus A = STOP \qquad hide - STOP$$

$$(x : C \to P(x))_{\Gamma(i/\gamma)} \setminus A = (x : C \to P(x)_{\Gamma(i/\gamma')}) \setminus A$$
$$= (x : C \to (P(x)_{\Gamma(i/\gamma')} \setminus A)) \quad \text{if} \quad A \cap C = \emptyset \qquad hide - step\ 1$$

$$(x : C \to P(x))_{\Gamma(i/\gamma)} \setminus A = (x : C \to P(x)_{\Gamma(i/\gamma')}) \setminus A$$
$$= \sqcap_{x \in C}(P(x)_{\Gamma(i/\gamma')} \setminus A) \quad \text{if} \quad C \subseteq A \qquad hide - step\ 2$$

$$SKIP \setminus A = SKIP \qquad hide - term$$

Figure 5.4: Laws for hiding

does not offer any action no local actions will be performed by the composition, as is the case in CSP. However, in CMVP process P can still execute interleaving call and/or return actions on its interface (independent on $STOP$) and so the parallel composition between P and $STOP$ is not equivalent to $STOP$ anymore. Consider for instance $P = a^1 \to P_{(1/a)}$, $P_{(1/a)} = b^1 \to P_{(1/\perp)}$, and $P_{(1/\perp)} = STOP$. We have traces$(P_\epsilon \| STOP) \neq \emptyset$; indeed, $a^1 b^1 \in$ traces$(P_\epsilon \| STOP)$. In fact in this example $(P_\epsilon \| STOP) = P_\epsilon$!

5.5 Hiding

Process P_Γ and $P_\Gamma \setminus A$ (where $A \subseteq \widetilde{\Sigma}$) will both perform the same events, the exception being that whenever P_Γ executes any event a in interface A then $P_\Gamma \setminus A$ will execute an internal action (τ). Hence, any event that is in the interface A cannot be picked up in the sequence of events of the trace, since it is hidden from the observer.

$$\text{traces}(P_\Gamma \setminus A) = \{tr \setminus A | tr \in \text{traces}(P_\Gamma)\}$$

Figure 5.4 enumerates the laws of hiding. In $hide{-}step\ 1$ and $hide{-}step\ 2$, γ and γ' both represents the stack content of the i-th stack before and after an execution (or transition):

if a local or an empty return event is executed then $\gamma' = \gamma$, else if a call or a non-empty return event is executed them $\gamma' \neq \gamma$ (in the execution of a call a symbol is pushed to the stack and with a return a symbol is popped from the stack), etc. Law $hide - combine$ states that interfaces hidden in succession and interfaces hidden at the same time (using the union operation) will always return the same outcome. Law $hide - STOP$ states that if there is no event and the hiding construct is applied, then nothing is hidden. Laws $hide - step$ 1 and $hide - step$ 2 describe instances of using the hiding construct over a prefix choice. In law $hide - step$ 1 none of the choice events belong to the hidden interface A, since there is no common event between interfaces A and C. As a result the same choice of events being offered. In law $hide - step$ 2 all of the choice events belong to the hidden interface A since interface C is a subset of A, resulting in the choice between the subsequent processes. Finally, law $hide - term$ states that hiding has no effect on the termination of a process.

5.6 Renaming

A process P_Γ and a forward renamed process $f(P)_\Gamma$ will behaves the same way, with the exception that $f(P)_\Gamma$ performs $f(a)$ whenever P_Γ would have performed a. The traces of process $f(P)_\Gamma$ are the same with process P_Γ with every event mapped through f.

$$\text{traces}(f(P_\Gamma)) = \{f(tr) | tr \in \text{traces}(P_\Gamma)\}$$

The backward renaming construct $f^{-1}(P_\Gamma)$ also behaves like process P_Γ, with the exception that any event a that is executed by $f^{-1}(P_\Gamma)$ corresponds to an event $f(a)$ executed by P_Γ. Hence, a trace tr of $f^{-1}(P_\Gamma)$, when mapped through the function f, will result in a trace $f(tr)$ of P_Γ.

$$\text{traces}(f^{-1}(P_\Gamma)) = \{tr | f(tr) \in \text{traces}(P_\Gamma)\}$$

Figure 5.5 enumerates the laws of renaming. In $f(.) - step$ 1, $f(.) - step$ 2, and Law $f^{-1}(.) - step$, γ and γ' both represents the stack content of the ith -stack before and after an

$$f(x : C \to P(x))_{\Gamma(i/\gamma)} = f(x : C \to P(x)_{\Gamma(i/\gamma')}) = y : f(C) \to f(P(f^{-1}(y)))_{\Gamma(i/\gamma')} \quad \text{if } f \text{ is 1-1} \qquad f(.) - step\ 1$$

$$f(x : C \to P(x))_{\Gamma(i/\gamma)} = f(x : C \to P(x)_{\Gamma(i/\gamma')}) = y : f(C) \to \sqcap_{x|f(x)=y} f(P(x))_{\Gamma(i/\gamma')} \qquad f(.) - step\ 2$$

$$f(SKIP) = SKIP \qquad f(.) - term$$

$$f^{-1}(x : C \to P(x))_{\Gamma(i/\gamma)} = f^{-1}(x : C \to P(x)_{\Gamma(i/\gamma')}) = y : f^{-1}(C) \to f^{-1}(P(f(y)))_{\Gamma(i/\gamma')} \qquad f^{-1}(.) - step$$

$$f^{-1}(SKIP) = SKIP \qquad f^{-1}(.) - term$$

Figure 5.5: Laws for Renaming

execution: if a local or an empty return event is executed then $\gamma' = \gamma$, else if a call or a non-empty return event is executed them $\gamma' \neq \gamma$ (in the execution of a call a symbol is pushed to the stack and with a return a symbol is popped from the stack). Law $f(.) - step\ 1$ states that if the mapping f is one-to-one then the renaming construct relates in an intuitive manner with prefix choice: a choice of events from interface C translates to a choice of events from $f(C) = \{f(c)|c \in C\}$. Function f is injective, hence any event y chosen will be mapped to exactly one event $x(= f^{-1}(y))$ from the original choice of events from interface C, so the subsequent actions are that of $P(x)_\Gamma$ transformed through the function f. Law $f(.) - step\ 2$ states that if a process is initially ready to execute any event from an interface C, then the initial choice for its renamed process is the set of events in interface $f(C)$. Unlike law $f(.) - step\ 1$, the mapping of function f is not one-to-one. Hence, any event y chosen will result in the execution of any of the processes which follow an event mapping to y: let a and b both appear in interface C, if function f has a mapping for both event a and c to an event c, then the renamed process executes the event c in two different ways, once resulting from an event a and once resulting from an event b. Finally, Both *term* laws state that both forward and backward renaming have no effect on the ability of a process

to terminate.

5.7 Sequential Composition

The sequential composition construct $P_\Gamma; Q_{\Gamma'}$ acts as process P_Γ until P_Γ successfully terminates, at which point control is then given to process $Q_{\Gamma'}$ to execute. The termination of process P_Γ does not mean the termination of the entire construct, instead process P_Γ's $\checkmark$ event occurs as an internal event τ. The traces of a sequential composition construct $P_\Gamma; Q_{\Gamma'}$ can be broken down into two parts: the traces of P_Γ before termination with its terminating trace, and the traces of $Q_{\Gamma'}$.

$$\begin{aligned} \text{traces}(P_\Gamma; Q_{\Gamma'}) \quad = \quad \{ \quad & tr | tr \in \text{traces}(P_\Gamma) \wedge \checkmark \notin \sigma(tr)\} \cup \{tr_1.tr_2 | tr_1\langle\checkmark\rangle \\ & \in \text{traces}(P_\Gamma) \wedge tr_2 \in \text{traces}(Q_{\Gamma'})\} \end{aligned}$$

Sequential composition satisfies all the laws stated in Figure 5.6. In Law ;-step, γ and γ' both represents the stack content of the i-th stack before and after an execution (or transition), as explained earlier. Law $;-assoc$ which is derived from the disjoint union operator states that a sequential composition construct is associative. The $;-unit$ laws provide a unit for the sequential composition construct. It states that if there is a sequential composition between *SKIP* (either as the left or right process) and another process P_Γ, the outcome will always be P_Γ. Law $;-step$ simply states that a prefix choice in a sequential composition construct is equal to a prefix choice of sequentially composed processes. Law $;-zero-l$ simply states that a sequential composition between process *STOP* (being the process on the left of the construct) and any other process, will always result in the process *STOP* since no events are initially offered and termination does not occur.

5.8 Interrupt

The interrupt construct $P_\Gamma \triangle Q_{\Gamma'}$ acts as P_Γ, but at any stage in its execution control can be passed to process $Q_{\Gamma'}$. Therefore the traces of the interrupt construct $P_\Gamma \triangle Q_{\Gamma'}$ can be broken down into two parts: the traces of P_Γ or else the non necessarily terminating traces

$$P_\Gamma; (Q_{\Gamma'}; R_{\Gamma''}) = (P_\Gamma; Q_{\Gamma'}); R_{\Gamma''} \qquad ; -assoc$$

$$SKIP; P_\Gamma = P_\Gamma \qquad ; -unit - l$$

$$P_\Gamma; SKIP = P_\Gamma \qquad ; -unit - r$$

$$(x : C \to P(x))_{\Gamma(i/\gamma)}; Q_{\Gamma'} = (x : C \to P(x)_{\Gamma(i/\gamma')}); Q_{\Gamma'}$$
$$= x : C \to (P(x)_{\Gamma(i/\gamma')}; Q_{\Gamma'}) \qquad ; -step$$

$$STOP; P_\Gamma = STOP \qquad ; -zero - l$$

Figure 5.6: Laws for sequential composition

of P_Γ concatenated with the traces of process $Q_{\Gamma'}$.

$$\text{traces}(P_\Gamma \triangle Q_{\Gamma'}) = \text{traces}(P_\Gamma) \cup \{tr_1.tr_2 | tr_1 \in \text{traces}(P_\Gamma) \wedge \checkmark \notin \sigma(tr_1) \wedge tr_2 \in \text{traces}(Q_{\Gamma'})\}$$

$$P_\Gamma \triangle (Q_{\Gamma'} \triangle R_{\Gamma''}) = (P_\Gamma \triangle Q_{\Gamma'}) \triangle R_{\Gamma''} \qquad \triangle - assoc$$

$$STOP \triangle P_\Gamma = P_\Gamma \qquad \triangle - unit - l$$

$$P_\Gamma \triangle STOP = P_\Gamma \qquad \triangle - unit - r$$

$$(x : C \to P(x))_{\Gamma(i/\gamma)} \triangle Q_{\Gamma'} = (x : C \to P(x)_{\Gamma(i/\gamma')}) \triangle Q_{\Gamma'}$$
$$= Q_{\Gamma'} \,\Box\, (x : C \to (P(x)_{\Gamma(i/\gamma')} \triangle Q_{\Gamma'})) \qquad \triangle - step$$

$$SKIP \triangle P_\Gamma = SKIP \,\Box\, P_\Gamma \qquad \triangle - term$$

Figure 5.7: Laws for interrupt

There are a number of laws appropriate for the Interrupt construct as given in Figure 5.7, governing its interaction with the choice construct and with termination. In $\triangle - step$, γ and γ' both represents the stack content of the i-th stack before and after an execution, as above. Law $\triangle - assoc$ which is derived from the disjoint union operator states that the interrupt construct is associative. Law $\triangle - step$ gives a description of how a prefix

choice interrupted by a process $Q_{\Gamma'}$ behaves: it either acts as the process $Q_{\Gamma'}$ instantly, or it executes an event in the prefix choice. If the latter is the case, the resulting process will also have to make the choice as mentioned above. The $\triangle - unit$ laws provide a unit for the interrupt construct. It states that if there is an interrupt between $STOP$ (either as the left or right process) and another process P_Γ, the outcome will always be P_Γ. Finally, $\triangle - term$ simply states that if termination occurs in the component process on the left side of the interrupt construct, then the interrupting process is discarded.

5.9 Recursion

In CSP and other finite-state process algebras a recursive process is simply described as a process that creates a loop from one state back to the same state and it is defined by a relation of form $P = F(P)$. If a CMVP recursive process is reasoned about in the same manner, so a CMVP recursive process will be defined by the relation $P_\Gamma = F(P_\Gamma)$. This however restrict the recursion construct in CMVP to a regular recursion construct. In the general sense (i.e. self-embedding recursion), the relation that defines a CMVP recursive process remains $P = F(P)$! We also note that a CMVP recursive process defines a loop from one *MVPDA state* back to the same MVPDA state; however, the content of the stack in operation is not required to be the same in the two instances where the MVPDA state is the same, but its behaviour should be decided instead by the executions that occur according to the recursive function F.

We therefore consider the recursive definition $P = F(P)$ within its proper place (as a CMVP process) i.e., $(P = F(P))_\Gamma$, or equivalently $P_\Gamma = F(P)_\Gamma$, where Γ is the set of stacks of the process in execution. Thus, the MVPDA state P with the set of stacks Γ behaves the same as the MVPDA state $F(P)$ with the same set of stacks. Hence, the traces(P_Γ) = traces$(F(P)_\Gamma)$. This recursive definition defines an *equation* which must be satisfied by the set traces(P_Γ). More precisely, traces(P_Γ) becomes a *fixed point* of the function on trace sets represented by the CMVP expression F; when the function is applied

to traces(P_Γ) to obtain traces($F(P)_\Gamma$), the result will again be traces(P_Γ).

Every CMVP process has the empty trace as one of its possible traces, hence $\langle\rangle \in$ traces(P_Γ); which means that, traces($STOP_\Gamma$) $\subseteq$ traces(P_Γ) $\Rightarrow$ traces($F(STOP)_\Gamma$) $\subseteq$ traces($F(P)_\Gamma$) = traces(P_Γ). This is established as a result of the *monotonic* nature of CMVP operators are in relation to $\subseteq$: hence, if traces(P_Γ) $\subseteq$ traces(Q_Γ), the traces($F(P)_\Gamma$) $\subseteq$ traces($F(Q)_\Gamma$) for any function F constructed out of CMVP operators and terms. Using standard induction it can be ascertained that for any n traces($F^n(STOP)_\Gamma$) $\subseteq$ traces($F(P)_\Gamma$) = traces(P_Γ) which points to the fact that all of the traces obtained by unwinding the definition $(P = F(P))_\Gamma$ n times are still traces of recursive process P_Γ. All of the $F^n(STOP)_\Gamma$ processes amounts to the finite unwinding of the recursive definition, so between them they account for all the possible traces of $(P = F(P))_\Gamma$. Hence

$$\text{traces}((P = F(P))_\Gamma) = \bigcup_{n \in \mathbb{N}} \text{traces}(F^n(STOP)_\Gamma)$$

.

5.10 Abstract

A CMVP process can contain several modules. The abstract construct extracts from a CMVP process only the internal trace of its first module, but then follows the rest of the original traces of the process. Only sub-modules that return to their parent modules are considered as complete sub-modules. Hence, unbalanced calls or returns cannot be in the trace of a complete sub-module since all complete traces of every complete sub-module are always balanced. An incomplete sub-module can happen at the end of a trace of a module, however it cannot have unbalanced returns in its trace. After the first module finishes its execution the abstract operator stops working, hence the rest of the traces of the process will not change:

$$\text{traces}(\overline{P_\Gamma}) = \{\mathfrak{A}(tr) | tr \in \text{traces}(P_\Gamma)\}$$

where $\mathfrak{A}(tr)$ is a function which extracts the trace tr' where tr' is the trace of $\overline{P_\Gamma}$, and tr is the trace of P_Γ. A definition for $\mathfrak{A}$ is given in Section 6.1.1.

Chapter 6

Trace Specification and Verification in CMVP

We assume that the visible partitions of events are always known by a CMVP trace observer. Hence, a CMVP trace observer can recognize when a system (MVPDA) executes a call or return event. This assumption enables the definition of four crucial functions in CMVP: abstract, stack extract, module extract, and completeness. These four functions allow our process algebra to specify certain properties that are useful in the system verification process. The abstract function is used to specify local properties of a module in a system. Stack limits, access control, and concurrent stack properties are specified using the stack extract function. The module extract function enables the specification of properties that are specific to one module of a system, while the completeness function is used to specify partial and total correctness of a system.

6.1 CMVP Trace Functions

6.1.1 Abstract Function

The *abstract function* or $\mathfrak{A}(tr)$ obtains the trace tr' of $\overline{P_\Gamma}$. Recall that for every trace tr' of $\overline{P_\Gamma}$ there is a trace tr of P_Γ. The definition of the abstract function is as follows: $\mathfrak{A}(tr) = \{l_0.{c_1}^i.{r_1}^i.l_1.{c_2}^i.{r_2}^i.l_2 \ldots {c_k}^i.{r_k}^i.l_k.w | \forall i : 1 \leq i \leq n \wedge tr = l_0.t_1.t_2 \ldots t_k.w \wedge l_0 \in \Sigma_l^* \wedge \forall x : 1 \leq x \leq k \wedge t_x \in \{{c_x}^i.s_x.{r_x}^i.l_x, \langle\rangle\} \wedge (s_x = \langle\rangle \vee \forall s' < s_x : (s' = \langle\rangle \vee |s'|_{\widetilde{\Sigma}_c^i} \geq$

$|s'|_{(A_r)_i}) \wedge |s_x|_{\widetilde{\Sigma}_c^i} = |s_x|_{\widetilde{\Sigma}_r^i}) \wedge {c_x}^i \in \widetilde{\Sigma}_c^i \wedge {r_x}^i \in \widetilde{\Sigma}_r^i \wedge l_x \in \Sigma_l^* \wedge (w = \langle\rangle \vee head(w) \in \widetilde{\Sigma}_r^{\checkmark} \vee \forall w' < w : (w' = \langle\rangle \vee |w'|_{\widetilde{\Sigma}_c^i} \geq |w'|_{\widetilde{\Sigma}_r^i}))\}$. The use of abstract function is illustrated in Section 6.2.4.

6.1.2 Stack Extract

In a CMVP trace the number of call events on a stack i in the set of stacks Γ, is equal to the number of stack symbols pushed onto the stack i, while the number of balanced return events in i is equal to the number of stack symbols popped off stack i. Hence, we define a function *stack extract* of a stack i or $\mathfrak{S}_i(tr)$ which will extract the stack content γ for a stack i in the set of stacks Γ from the CMVP trace tr.

$\mathfrak{S}_i(tr)$ of a process $P_{(\gamma_1,...,\gamma_i,...,\gamma_n)}$ is defined as follows (with $\mathfrak{R}(tr)$ denoting the reversal of the trace tr): $\mathfrak{S}_i(tr) = \{{c_{i+j}}^i.{c_{i+j-1}}^i \ldots {c_{i+2}}^i.{c_{i+1}}^i.\bot | tr' = tr \setminus \widetilde{\Sigma}_l^{\checkmark} \wedge sq = \mathfrak{R}(tr').\bot \wedge sq = {s_{i+j+1}.c_{i+j}}^i.s_{i+j}.{c_{i+j-1}}^i \ldots s_{i+2}.{c_{i+1}}^i.s_{i+1}.{r_i}^i.s_i.{r_{i-1}}^i \ldots s_3.{r_2}^i.s_2.{r_1}^i.s_1.\bot \wedge \forall x : 1 \leq x \leq i \wedge {r_x}^i \in \{\Sigma_r^i \cup \langle\rangle\} \wedge \forall y : 1 \leq y \leq j \wedge {c_{i+y}}^i \in \{\Sigma_c^i \cup \langle\rangle\} \wedge \forall z : 1 \leq z \leq i+j+1 \wedge s_z \in S^* \wedge S = \{s | \forall s' < s : (s' = \langle\rangle \vee |s'|_{\Sigma_r^i} \geq |s'|_{\Sigma_c^i}) \wedge |s|_{\Sigma_r^i} = |s|_{\Sigma_c^i}\}\}$. The use of stack extract is illustrated in Sections 6.2.1 and 6.2.2.

6.1.3 Module Extract

Module extract or $\mathfrak{M}(tr, a^i)$ extracts from the trace tr the trace tr' of the module which starts with the call event a^i. $\mathfrak{M}(tr)$ extracts the trace tr' of the first module from the trace tr: $\mathfrak{M}(tr) = \{tr' | tr = tr'.tr'' \wedge \forall t < tr' : (t = \langle\rangle \vee |t|_{\Sigma_c^1} \geq |t|_{\Sigma_r^1}) \wedge (tr'' = \langle\rangle \vee tr'' = \langle\checkmark\rangle \vee (head(tr'') \in \widetilde{\Sigma}_r \wedge |tr'|_{\widetilde{\Sigma}_c} = |tr'|_{\widetilde{\Sigma}_r}))\}$ and $\mathfrak{M}(tr, a^i) = \{tr' | tr = tr''.a^i.tr''' \wedge |tr''|_{a^i} = 0 \wedge tr' = \mathfrak{M}(tr''')\}$. The use of module extract is illustrated and made clearer in Section 6.2.4 and in Section 6.2.5.

6.1.4 Completeness

Completeness or $\mathfrak{C}(tr, a^i)$ is a function which verifies that a trace tr contains the complete trace of a sub-module. Hence, if a sub-module is invoked by call event a^i, then tr must include the call event a^i and its balanced return. It will return the desired trace if it is

in tr, else it will return the empty trace: $\mathfrak{C}(tr, a^i) = \{tr' | (tr = tr''.tr'.tr''' \wedge head(tr') = a^i \wedge foot(tr') \in \widetilde{\Sigma}_r^i \wedge t < tr' : (t = \langle\rangle \vee |t|_{\widetilde{\Sigma}_c^i} \geq |t|_{\widetilde{\Sigma}_r^i}) \wedge |tr'|_{\widetilde{\Sigma}_c^i} = |tr'|_{\widetilde{\Sigma}_r^i} \wedge |tr''|_{a^i} = 0) \vee tr' = \langle\rangle\}$. The use of completeness will be illustrated in Section 6.2.5.

6.2 CMVP Trace Specification

CMVP encompasses CSP and it behaves exactly like CSP when all the executing events are locals. Hence, CMVP can specify any property of a system that CSP can. CMVP can also specifies various crucial properties for software verification that regular or a context-free process algebra are unable to specify as outlined below:

6.2.1 Access Control

Unlike any regular or a context-free process algebra, CMVP can specify the access control properties of a module. It can specify that a module can be invoked if a certain property holds. For instance, a specification might require that a procedure A (called by a call event a^i) can be invoked only if another procedure B (called by a call event b^i) is on the stack i in the set of stacks Γ. In CMVP, this property is expressed as $S(tr) = |\mathfrak{S}_i(tr)|_{b^i} \neq 0 \Rightarrow |\mathfrak{S}_i(tr)|_{a^i} \neq 0$.

6.2.2 Stack Limit

CMVP can specify that a property holds, whenever the size of the i-th stack of a CMVP process is bound by a given constant. For instance, a specification might require that there will be no occurrence of an event a in the trace, if the size of the i-th stack is less than 7. In CMVP, this predicate can be expressed as $S(tr) = |\mathfrak{S}_i(tr)| < 7 \Rightarrow |tr|_a = 0$.

6.2.3 Concurrent Stack Properties

In a parallel composition construct a concurrent stack property is a requirement that a property holds in some stack i of a process $P_{(1,\ldots,n)}$ and another property holds in some stack $n+i$ of a concurrent process $Q_{(n+1,\ldots,n+n)}$. Unlike any regular or context-free process

algebra, CMVP is able to specify this predicate. From the trace of $P_{\Gamma}{}_{A}\|_{B}Q_{\Gamma'}$ one can specify that if one stack property p holds in $P_{(\gamma_1,...,\gamma_n)}$, then another stack property q holds in $Q_{(\gamma_{n+1},...\gamma_{n+m})}$. If tr is the trace of $P_{\Gamma}{}_{A}\|_{B}Q_{\Gamma'}$, then $tr \upharpoonright A^{\checkmark}$ is the trace of P_{Γ} and $tr \upharpoonright B^{\checkmark}$ is the trace of $Q_{\Gamma'}$. Indeed, one can use the fact that $(\mathfrak{S}_i(tr) \upharpoonright A^{\checkmark})$ is the stack i of P_{Γ} and $(\mathfrak{S}_{(n+i)}(tr) \upharpoonright B^{\checkmark})$ is the stack $n+i$ of Q_{Γ}. For instance, a specification might require that if the stack size is less than 7 in stack i of process $P_{(\gamma_1,...,\gamma_n)}$, then there will be no invocation for module A (called by a call event a^{n+i}) in $Q_{\Gamma'}$. In CMVP, this property can be expressed as $S(tr) = |(\mathfrak{S}_i(tr) \upharpoonright A^{\checkmark})| < 7 \Rightarrow |(\mathfrak{S}_{(n+i)}(tr) \upharpoonright B^{\checkmark})|_{a^{n+i}} = 0$, where tr is the trace of $P_{\Gamma}{}_{A}\|_{B}Q_{\Gamma'}$.

6.2.4 Internal Properties of a Module

CMVP allows the internal trace of a (possibly recursive) module to be extracted from a CMVP trace and any trace properties can then be specified on the extracted trace. Let A be a recursive module (which is called by call event a^i) in process $P_{(\gamma_1,...,\gamma_n)}$. The trace of module A is extracted using $\mathfrak{M}(tr, a^i)$, where tr is the trace of $P_{(\gamma_1,...,\gamma_n)}$. Using the abstract function, the internal trace can be further extracted: $\mathfrak{A}(\mathfrak{M}(tr, a^i))$ is the internal trace of module A. For instance, a specification might require that the number of b events is always larger or equal than the number of c events in the local execution of A. this property can be expressed as $S(tr) = |\mathfrak{A}(\mathfrak{M}(tr, a^i))|_b \geq |\mathfrak{A}(\mathfrak{M}(tr, a^i))|_c$.

6.2.5 Pre- and Post-Conditions

CMVP can accommodate the specification of the pre- and post-conditions of a module. Hence, partial and total correctness can be specified in CMVP. The partial correctness of a procedure A specifies that if the pre-condition p holds when the procedure A is called, then if the procedure terminates the post-condition q is satisfied upon the return of A. Let module A be called by a^i. During the invocation of A, if some event b always precedes another event c, if A returns, then the number of b events will be smaller than the number of c events in module A. This predicate is specified as: $S(tr) = (tr = tr_1.a^i.tr_2) \wedge (tr_1 \upharpoonright$

$b = \langle\rangle \Rightarrow tr_1 \upharpoonright c = \langle\rangle) \Rightarrow \mathfrak{C}(a^i.tr2, a^i) = \langle\rangle \vee (\mathfrak{C}(a^i.tr2, a^i) \neq \langle\rangle \wedge |\mathfrak{M}(tr2)|_b < |\mathfrak{M}(tr2)|_c)$. Similarly, the total correctness of a procedure A specifies that if the pre-condition p holds when the procedure is called, then the procedure must terminate and the post-condition q must be satisfied upon the return of the procedure A. For instance, a specification might require that during the invocation of A, if some event b always precede another event c, then A returns and the number of b events will be smaller than the number of c events in module A. The property can be specified as: $S(tr) = (tr = tr_1.a^i.tr_2) \wedge (tr_1 \upharpoonright b = \langle\rangle \Rightarrow tr_1 \upharpoonright c = \langle\rangle) \Rightarrow \mathfrak{C}(a^i.tr2, a^i) = \langle\rangle \wedge (\mathfrak{C}(a^i.tr2, a^i) \neq \langle\rangle \wedge |\mathfrak{M}(tr2)|_b < |\mathfrak{M}(tr2)|_c)$.

6.3 CMVP Trace Verification

Using CMVP various non-regular specification properties can be verified, including the properties stated in Section 6.2. CMVP verification shares many features of CSP verification. The major difference between CMVP and CSP verification rules is the presence of a set of stacks in a CMVP process. Since CMVP processes are MVPDAs, CMVP verification rules are applied over a visible alphabet, whereas the CSP verification rules are applied over a local alphabet.

6.3.1 Prefix Choice

The CMVP prefix choice construct contains a number of component processes. It is always prepared to execute any one of the menu of events offered. The antecedent to the rule assumes a family of specifications $S^a(tr)$, one for each of the components $P(a)_{\Gamma(i/\gamma')}$ where $(x : A \to P(x))_{\Gamma(i/\gamma)} = x : A \to P(x)_{\Gamma(i/\gamma')}$. The proof rule is:

$$\frac{\forall a \in A : P(a)_{\Gamma(i/\gamma')} \ \vdash \ S^a(tr)}{(x : A \to P(x))_{\Gamma(i/\gamma)} \ \vdash \ tr = \langle\rangle \vee \exists a \in A : head(tr) = a \wedge S^a(tail(tr))}$$

6.3.2 Choice

The choice constructs $P_\Gamma \,\Box\, Q_{\Gamma'}$ or $P_\Gamma \sqcap Q_{\Gamma'}$ acts either as P_Γ or as $Q_{\Gamma'}$. If $P_\Gamma \vdash S(tr)$ and $Q_{\Gamma'} \vdash T(tr)$, then the choice construct $P_\Gamma \,\Box\, Q_{\Gamma'}$ or $P_\Gamma \sqcap Q_{\Gamma'}$ satisfies the disjunction of

these two specifications:

$$\frac{P_\Gamma \vdash S(tr) \quad Q_{\Gamma'} \vdash T(tr)}{P_\Gamma \,\square\, Q_{\Gamma'} \vdash S(tr) \vee T(tr)}$$

and

$$\frac{P_\Gamma \vdash S(tr) \quad Q_{\Gamma'} \vdash T(tr)}{P_\Gamma \sqcap Q_{\Gamma'} \vdash S(tr) \vee T(tr)}.$$

Let P_ϵ and Q_ϵ be two CMVP processes. Let A be a module which can execute only event d, called by c^i and returned by e^i. Let B be a module which after executing an event b executes its sub-module A. C is another module which first executes its sub-module A then executes an event h. Process P_ϵ calls module B and ends its execution when B returns, while Q_ϵ calls module C and ends its execution when C returns: $P = a^i \rightarrow P1_{(i/a)}$, $P1 = b \rightarrow P2$, $P2 = c^i \rightarrow P3_{(i/ca)}$, $P3 = d \rightarrow P4$, $P4_{(i/ca)} = e^i \rightarrow P5_{(i/a)}$, $P5_{(i/a)} = f^i \rightarrow P6_{(i/\perp)}$, $P6 = STOP$, and $Q = g^j \rightarrow Q1_{(j/g)}$, $Q1 = c^j \rightarrow Q2_{(j/c)}$, $Q2 = d \rightarrow Q3$, $Q3_{(j/cg)} = e^j \rightarrow Q4_{(j/g)}$, $Q4 = h \rightarrow Q5$, $Q5_{(j/g)} = i^j \rightarrow Q6_{(j/\perp)}$, $Q6 = STOP$.

So $\text{traces}(P_\epsilon) = \{\langle\rangle, \langle a^i\rangle, \langle a^i, b\rangle, \langle a^i, b, c^i\rangle, \langle a^i, b, c^i, d\rangle, \langle a^i, b, c^i, d, e^i\rangle, \langle a^i, b, c^i, d, e^i, f^i\rangle\}$, $\text{traces}(Q_\epsilon) = \{\langle\rangle, \langle g^j\rangle, \langle g^j, c^j\rangle, \langle g^j, c^j, d\rangle, \langle g^j, c^j, d, e^j\rangle, \langle g^j, c^j, d, e^j, h\rangle, \langle g^j, c^j, d, e^j, h, i^j\rangle\}$, and $\text{traces}(P_\epsilon \,\square\, Q_\epsilon) = \text{traces}(P_\epsilon \sqcap Q_\epsilon) = \{\langle\rangle, \langle a^i\rangle, \langle g^j\rangle, \langle a^i, b\rangle, \langle g^j, c^j\rangle, \langle a^i, b, c^i\rangle, \langle g^j, c^j, d\rangle, \langle a^i, b, c^i, d\rangle, \langle g^j, c^j, d, e^j\rangle, \langle a^i, b, c^i, d, e^i\rangle, \langle g^j, c^j, d, e^j, h\rangle, \langle a^i, b, c^i, d, e^i, f^i\rangle, \langle g^j, c^j, d, e^j, h, i^j\rangle\}$. P_ϵ satisfies the following non-regular property: module A can be called only if a is on its i-th stack, while Q_ϵ satisfies another non-regular property: module A can be called only if g is on its j-th stack:

$$P_\epsilon \vdash S(tr) = (|\mathfrak{S}_i(tr)|_{a^i} = 0 \Rightarrow |\mathfrak{S}_i(tr)|_{c^i} = 0)$$

$$Q_\epsilon \vdash T(tr) = (|\mathfrak{S}_j(tr)|_{g^j} = 0 \Rightarrow |\mathfrak{S}_j(tr)|_{c^j} = 0)$$

Then $P_\epsilon \,\square\, Q_\epsilon$ or $P_\epsilon \sqcap Q_\epsilon$ meets the specification

$$(|\mathfrak{S}_i(tr)|_{a^i} = 0 \Rightarrow |\mathfrak{S}_i(tr)|_{c^i} = 0) \vee (|\mathfrak{S}_j(tr)|_{g^j} = 0 \Rightarrow |\mathfrak{S}_j(tr)|_{c^j} = 0)$$

6.3.3 Parallel Composition

A trace tr of the a parallel composition construct $(P_A \|_B Q)_{\Gamma.\Gamma'}$ will execute events from P_Γ and execute events from $Q_{\Gamma'}$, contained within the alphabets $A^\checkmark$ and $B^\checkmark$, respectively. Hence, if $P_\Gamma \vdash S(tr)$, then $S(tr) \upharpoonright A^\checkmark$ must hold. Also, if $Q_{\Gamma'} \vdash T(tr)$, then $T(tr) \upharpoonright B^\checkmark$ must hold. Finally, only events in $A^\checkmark$ or $B^\checkmark$ can be executed the parallel composition, hence $\sigma(tr) \subseteq (A \cup B)^\checkmark$. The proof rule is as follows:

$$\frac{\begin{array}{c} P_\Gamma \ \vdash \ S(tr) \\ Q_{\Gamma'} \ \vdash \ T(tr) \end{array}}{(P_A \|_B Q)_{\Gamma.\Gamma'} \ \vdash \ S(tr \upharpoonright A^\checkmark) \wedge T(tr \upharpoonright B^\checkmark) \wedge \sigma(tr) \subseteq (A \cup B)^\checkmark}.$$

Intuitively, parallel composition corresponds to conjunction: both the constraints S and T hold in the parallel composition (on their respective interfaces).

The parallel composition $(P_{\{a^i, b, c^i, d, e^i, f^i\}} \|_{\{g^{n+i}, c^{n+i}, d, e^{n+i}, h, i^{n+i}\}} Q)_{((\gamma_1,\ldots,\gamma_n)\cdot(\gamma_{n+1},\ldots,\gamma_{n+m}))}$ between the two processes P_ϵ and Q_ϵ from the Section 6.3.2 will satisfy

$$\begin{array}{l} S(tr \upharpoonright \{a^i, b, c^i, d, e^i, f^i\}^\checkmark) \wedge T(tr \upharpoonright \{g^{n+i}, c^{n+i}, d, e^{n+i}, h, i^{n+i}\}^\checkmark) \\ \wedge \sigma(tr) \subseteq \{a^i, b, c^i, d, e^i, f^i, g^{n+i}, c^{n+i}, e^{n+i}, h, i^{n+i}\}^\checkmark \end{array}$$

which reduces to

$$\begin{array}{l} |\mathfrak{S}(tr)_{(i/\gamma)}|_{a^i} \wedge |\mathfrak{S}(tr)_{(n+i/\delta)}|_{g^{n+i}} = 0 \\ \Rightarrow |\mathfrak{S}(tr)_{(i/\gamma)}|_{c^i} = 0 \wedge \sigma(tr) \subseteq \{a^i, b, c^i, d, e^i, f^i, g^{n+i}, c^{n+i}, e^{n+i}, h, i^{n+i}\}^\checkmark \end{array}$$

6.3.4 Hiding

A trace of the process $P_\Gamma \setminus A$ is simply a trace of P_Γ less all the events in the interface A. Therefore, for every trace of $P_\Gamma \setminus A$ there is a corresponding trace of P_Γ. The following is the inference rule:

$$\frac{P_\Gamma \ \vdash \ S(tr)}{P_\Gamma \setminus A \ \vdash \ \exists tr_1 : tr_1 \setminus A = tr \wedge S(tr_1)}.$$

The process P_ϵ of Section 6.3.2 meets the non-regular specification that there will be no occurrence of event d in the internal trace of the module which is invoked by call event a^i

$$P_\epsilon \vdash S(tr) = (tr = tr'.a^i.tr'' \Rightarrow |\mathfrak{A}(tr'')|_d = 0)$$

Hence, the process $P_\epsilon \setminus \{c^i\}$ meets the following specification:

$$S'(tr) = (\exists tr_1 : tr_1 \setminus \{c^i\} = tr \wedge (tr_1 = tr'.a^i.tr'' \Rightarrow |\mathfrak{A}(tr'')|_d = 0))$$

6.3.5 Abstract

A trace of the abstract construct $\overline{P_\Gamma}$ is obtained from the trace of P_Γ by removing all the traces of the sub-modules of the top level module. Hence, the inference rule is as follows:

$$\frac{P_\Gamma \ \vdash \ S(tr)}{\overline{P_\Gamma} \ \vdash \ \exists tr_1 : \mathfrak{A}(tr_1) = tr \wedge S(tr_1)}$$

.

The process P_ϵ of Section 6.3.2 satisfies the partial correctness property that if b is in the trace when a module is called, if the module returns then there will be a d in the trace:

$$P_\epsilon \vdash S(tr) = ((tr = tr'.a^i.tr'' \wedge |tr'|_b \neq 0 \wedge a^i \in \widetilde{\Sigma}_c^i) \Rightarrow (\mathfrak{C}(a^i.tr'', a^i) = \langle\rangle \vee tr_1.|\mathfrak{M}(tr'')|_d \neq 0))$$

so $\overline{P_\epsilon}$ will satisfy

$$S'(tr) = \exists tr_1 : \mathfrak{A}(tr_1) = tr \wedge (\quad (tr = tr'.a^i.tr'' \wedge |tr'|_b \neq 0 \wedge a^i \in \widetilde{\Sigma}_c^i) \Rightarrow (\mathfrak{C}(a^i.tr'', a^i) = \langle\rangle \vee tr_1.|\mathfrak{M}(tr'')|_d \neq 0) \quad)$$

6.3.6 Renaming

The trace tr of a renamed process $f(P_\Gamma)$ will correspond to a renamed trace $f(tr_1)$ for some tr_1 of P_Γ. The inference rule for translating specifications through a forward renaming is as follows:

$$\frac{P_\Gamma \ \vdash \ S(tr)}{f(P_\Gamma) \ \vdash \ \exists tr_1 : S(tr_1) \wedge f(tr_1) = tr}$$

A particular specification S can be translated through the renaming function f to a specification R. This will hold if $R(tr)$ can be shown to translate to S correctly: $\forall tr : (S(tr) \Rightarrow R(f(tr)))$. If tr is a trace of $f^{-1}(P_\Gamma)$, then $f(tr)$ is a trace of P_Γ, so it must satisfy whatever specification P_Γ is known to satisfy. The inference rule is as follows:

$$\frac{P_\Gamma \ \vdash \ S(tr)}{f^{-1}(P_\Gamma) \ \vdash \ S(f(tr))}$$

6.3.7 Sequential Composition

The sequential composition construct $P_\Gamma; Q_{\Gamma'}$ acts as P_Γ until P_Γ eventually terminates, after which it acts as $Q_{\Gamma'}$. Any given trace of process $P_\Gamma; Q_{\Gamma'}$ will have one of the two alternatives: it is either a trace of P_Γ that has yet to perform termination, else it is a trace of P_Γ then followed by a trace of $Q_{\Gamma'}$. The proof rule is as follows:

$$\frac{\begin{array}{c} P_\Gamma \ \vdash \ S(tr) \\ Q_{\Gamma'} \ \vdash \ T(tr) \end{array}}{P_\Gamma; Q_{\Gamma'} \ \vdash \ \neg term(tr) \wedge S(tr) \vee \exists tr_1, tr_2 : tr = tr_1 tr_2 \wedge S(tr_1\langle\checkmark\rangle) \wedge T(tr_2)}$$

where $term(tr) = \checkmark \in \sigma(tr)$ denotes that the trace corresponds to a terminating execution.

6.3.8 Interrupt

Any given trace of the interrupt construct $P_\Gamma \triangle Q_{\Gamma'}$ is either a trace of P_Γ (which has successfully terminated) or it is a non-terminated trace of P_Γ followed by a trace of $Q_{\Gamma'}$. The inference rule is as follows:

$$\frac{\begin{array}{c} P_\Gamma \ \vdash \ S(tr) \\ Q_{\Gamma'} \ \vdash \ T(tr) \end{array}}{P_\Gamma \triangle Q_{\Gamma'} \ \vdash \ S(tr) \vee \exists tr_1, tr_2 : tr = tr_1 tr_2 \wedge \neg term(tr_1) \wedge S(tr_1) \wedge T(tr_2)}.$$

6.3.9 Recursion

A recursive process N_Γ is recursively defined by the equation $(N = P)_\Gamma$ or equivalently $(N = F(N))_\Gamma$. Hence a rule which is sufficient to define that $N_\Gamma \vdash S(tr)$ is as follows:

$$\frac{\forall Y_\Gamma : (Y_\Gamma \ \vdash \ S(tr) \Rightarrow F(Y)_\Gamma \ \vdash \ S(tr))}{N_\Gamma \ \vdash \ S(tr)} \ \big[S(\langle\rangle)\big]$$

This rule covers all the aspects necessary for defining by induction that $N_\Gamma \vdash S(tr)$, hence it is valid . The traces of N_Γ are those of $\bigcup_{i\in\mathbb{N}} \text{traces}((F^i(STOP))_\Gamma)$, and all the finite unwindings of $(F(Y))_\Gamma$ begin from the process $STOP$. The inductive hypothesis is that $(F^i(STOP))_\Gamma \vdash S(tr)$. The side conditions $S(\langle\rangle)$ corresponds to the base case, since it is equivalent to $STOP \vdash S(tr)$, which is equivalent to $(F^0(STOP))_\Gamma \vdash S(tr)$.

Chapter 7

Conclusions

CMVP defines a superset of CSP by merging the interesting properties of finite-state algebras (provided by CSP) with the context-free features of MVPL. MVPDA on its own cannot work in this context, but by using two extra constructions (disjoint operations over MVPDA and stack renaming) over the original definition closure under interesting operations is attained. The closure of MVPDA under these operations has paved the way for an MVPDA-based process algebra that can be used to specify and verify concurrent and recursive systems. Since LTS semantics serve as the underlying semantic model for any process algebra, LTS semantics corresponding to an MVPDA has also been provided to further strengthen the definition of CMVP.

Although CSP and other finite-state process algebras have proven useful for the specification and verification of hardware, communication protocols, and drivers, they are not able to adequately specify and verify more complex application software. Indeed, application software has a large number of distinct finite states, and so finite-state mechanisms are no longer practical in this context. CMVP therefore opens up a new domain in formal methods (and more specifically algebraic methods such as model-based testing) for the specification and verification of application software. Since it is an infinite-state process algebra, its context-free features prove adequate for specifying and verifying these complex systems. We thus present CMVP, the first fully compositional multi-stack visibly pushdown

process algebra, as a superset of CSP. The definition of the semantics of MVPDA in terms of labelled transition systems establishes MVPL as the domain language for CMVP. Also, the operational and trace semantics for CMVP are derived from the semantics of MVPDA. Evidently, the closure of CMVP under all its operators offers prove that it is a process algebra.

We also define functions on CMVP traces that extract stack and module information from CMVP traces. Using these functions, useful properties for software verification such as the access control of a module, stack limits, concurrent stack properties, internal properties of a module, and pre-/post-conditions of a module, which context-free or regular process algebras are unable to specify can be specified in CMVP. This dissertation has thus put down the foundation that will enable MVPDA (rather than finite automata) to be the basis of concurrent process algebraic tools and theories, hence allowing for the formal specification and verification of application software.

7.1 Advantages of CVP over Other Process Algebrae

The domain language of CMVP (MVPL) encompasses the spectrum of regular languages and dives into the context-free spectrum. Therefore, unlike finite-state process algebra CMVP naturally models recursive modules. Also CMVP combines regular language features with context-free features, hence it can model multi-threaded modules just as effectively as finite-state process algebra. It has the advantage of being able to represent both recursive and multi-threaded modules compositionally, unlike any other process algebra.

Another advantage of CMVP over other process algebras is that CMVP processes operate over visible symbols, hence the stack content of a process can easily be identified by observing the process. Therefore, the specification and verification of (concurrent) stack properties is possible. An instance of this is a specification that requires that a module A can execute only after a module B has executed, with no interleaving call to an overriding module C [2]. Unlike other context free process algebra, CMVP is able to offer this feature

as a result of the visible nature of its domain language (MVPL).

The environment of a CMVP process is also aware when a module executes (by performing a call event) and when the executed module terminates (by performing a return action) since its domain language uses visible alphabets. Hence, the specification and verification of both pre-conditions and post-conditions of recursive modules is also possible. The pre-conditions of a module can be verified at the beginning of its execution, while its post-conditions is verified at the end point.

Another benefit of CMVP over other process algebras is the use of the newly introduced abstract operator that allows for the abstraction of modules from the process environment. Variants of the CMVP abstract operator can be easily defined; for instance a variant that terminates a process when the abstracted module terminates or a variant that hides only a specific sub-module can be created.

Finally, the CMVP module extract function gives CMVP an edge over other process algebras. It allow the trace of a module to be easily extracted from the trace of a process. By combining the module extract and abstract function of CMVP the internal trace of a module can be identified. Hence with CMVP the specification and verification of the internal properties of modules is possible, giving it an edge over other process algebras.

7.2 Future

This dissertation has laid out the basis for MVPDA-based axiomatic verification. As would be required for any process algebra, a preliminary proof system based on the trace model has been defined. MVPDA-base failures, divergences and infinite (FDI) traces model, pre-order relations, equivalence testing are all areas that are open for future research. CMVP opens up a whole new realm that allows for the specification and verification of real-time and concurrent systems using MVPDA-based axiomatic verification. It also offers important features that other process algebras cannot as enumerated in Section 7.1. This leads us to conclude that the future will be dominated by our MVPDA-based process algebra rather

than context free and regular process algebras.

Bibliography

[1] R. Alur, M. Arenas, P. Barcelo, K. Etessami, N. Immerman, and L. Libkin. First-order and temporal logics for nested words. In *Proceedings of the 22nd IEEE Symposium on Logic in Computer Science*, pages 151–160. IEEE Computer Society, 2007.

[2] R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 04)*, volume 2988 of *Lecture Notes in Computer Science*, pages 467–481. Springer, 2004.

[3] R. Alur and P. Madhusudan. Visibly pushdown languages. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC 04)*, pages 202–211. ACM Press, 2004.

[4] J. C. M. Baeten and W. P. Weijland. *Process Algebra*. Cambridge University Press, 1990.

[5] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, 1985.

[6] J. A. Bergstra and J. W. Klop. Process theory based on bisimulation semantics. In J. W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 50–122. Springer, 1988.

[7] S. D. Brookes, A. W. Roscoe, and D. J. Walker. An operational semantics for CSP. *Technical Report*, 1988.

[8] S.D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, 1984.

[9] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems: Advanced Lectures*, volume 3472 of *Lecture Notes in Computer Science*. Springer, 2005.

[10] S. D. Bruda. Preorder relations. In Broy et al. [9], pages 117–149.

[11] Stefan D Bruda and Md Tawhid Bin Waez. Unrestricted and disjoint operations over multi-stack visibly pushdown languages. In *Proceedings of the 6th International Conference on Software and Data Technologies (ICSOFT 2011)*, pages 98–103, Seville, Spain, July 2011.

[12] Dario Carotenuto, Aniello Murano, and Adriano Peron. 2-visibly pushdown automata. In *Developments in Language Theory*, pages 132–144. Springer, 2007.

[13] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[14] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19*, pages 19–32, Providence, 1967. American Mathematical Society.

[15] A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1988.

[16] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, pages 576–580, 1969.

[17] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[18] S La Torre, P Madhusudan, and G Parlato. 2-vpas are not determinizable. http://www.cs.uiuc.edu/ madhu/vpa/wrong-proof-CMP07.html, 2007.

[19] Salvatore La Torre, Parthasarathy Madhusudan, and Gennaro Parlato. A robust class of context-sensitive languages. In *Logic in Computer Science, 2007. LICS 2007. 22nd Annual IEEE Symposium on*, pages 161–170. IEEE, 2007.

[20] Salvatore La Torre, Parthasarathy Madhusudan, and Gennaro Parlato. The language theory of bounded context-switching. In *LATIN 2010: Theoretical Informatics*, pages 96–107. Springer, 2010.

[21] J. Mccarthy. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, pages 33–70. North-Holland, 1963.

[22] A. J. R. G. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[23] R. Milner. A complete inference system for a class of regular behaviours. *Journal of Computer and System Sciences*, 28:439–466, 1984.

[24] S. Schneider. *Concurrent and Real Time Systems: The CSP Approach*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[25] D. Scott and C. Strachey. Towards a mathematical semantics for computer languages. In J. Fox, editor, *Computers and Automata*, pages 19–46. John Wiley, 1972.

[26] Jiří Srba. Beyond language equivalence on visibly pushdown automata. *arXiv preprint arXiv:0901.2068*, 2009.

[27] Md Tawhid Bin Waez. Communicating visibly pushdown processes. Master's thesis, Bishop's University, 2008.