COMMUNICATING VISIBLY PUSHDOWN PROCESSES

by

MD TAWHID BIN WAEZ

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
December 2008

Abstract

Visibly pushdown languages are a subclass of context-free languages that is closed under all the useful operations, namely union, intersection, complementation, renaming, concatenation, prefix, and Kleene star. The existence of a concurrent, fully compositional process algebra based on such languages requires that these languages be also closed under two more operations, namely shuffle, and hiding. We prove here both of these closure properties. We also give the semantics of visibly pushdown automata in terms of labelled transition systems.

We then propose Communicating Visibly pushdown Processes (CVP), a fully compositional concurrent process algebra based on visibly pushdown automata. CVP is a superset of CSP, thus combining all the good properties of finite-state algebrae with context-free features. Unlike any other process algebra, CVP includes support for parallel composition but also for self-embedding recursion.

We present the syntax, operational semantics, trace semantics, trace specification, and trace verification of CVP. A CVP trace observer can extract stack and module information from the trace; as a result one can specify and verify many software properties which cannot be specified in any other existing process algebra. Such properties include the access control of a module, stack limits, concurrent stack properties, internal property of a module, pre-/post-conditions of a module, etc. CVP lays the basis of algebraic conformance testing for infinite-state processes, such as application software.

Acknowledgments

First of all I want to thank the almighty God. I especially want to thank my supervisor, Stefan D. Bruda, for his guidance during my research and study at Bishop's University. His perpetual energy and enthusiasm in research had motivated me. In addition, he was always accessible and willing to help in my research. He was always beside me for any kind of issues. As a result, research life became smooth and rewarding for me. His deep knowledge in formal language and keen interest in conformance testing make this research possible.

I want to thank the members of my defense committee Jurgen Dingel and Madjid Allili. This manuscript has been enhanced by their thoughtful comments.

I also want to thank my family, friends, and well-wishers. There is no way to repay the contribution of a teacher or a parent, so I am always grateful to all of my teachers and parents.

Contents

List of Figures

Chapter 1

Introduction

Pushdown automata naturally model the control flow of sequential computation in typical programming languages with nested, potentially recursive invocations of program modules such as procedures and methods. Many non-regular properties are therefore required for software verification; such that inspection of the stack, or matching of calls and returns, total/partial correctness of a module, etc. Such properties generate an infinite state space, which cannot be handled by finite-state verification techniques such as finite state *model checking* [29], finite state *process algebrae* [13], etc. Most of the contemporary software use many parallel components (such as multiple threads). In addition, many conformance-testing[1] techniques (such as may/must testing [30]) use test cases that run in parallel with the process under test. Recursive concurrency is therefore required for software verification, but cannot be provided by context-free verification techniques (such as basic process algebra or BPA [18]) since context-free languages are not closed under intersection [10], or by finite state verification techniques as they cannot support recursive modules.

1.1 Concurrent Process Algebra

A process algebra represents a mathematically rigourous framework for modeling concurrent systems of interacting processes. The process-algebraic approach relies on equational and inequational reasoning as the basis for analyzing the behavior of a system. Two separate works by Hoare [35]

[1] By conformance-testing we mean here any formal method that determines whether a system meets a specified standard.

and Milner [41] are marked as the origin of the process algebrae, which have been an active area of research since. In particular, researchers have developed a number of different process-algebraic theories in order to capture different aspects of system behavior; each such formalism generally includes the following semantic approaches:

Operational semantics: The behavior of a system is modeled as an execution of an abstract machine consisting of only a set of states and a set of transitions [20, 40].

Denotational semantics: More abstract than operational semantics, system behaviors are usually modeled by a function transforming input into output [50]. Denotational semantics can introduce behavioral equivalences (e.g. refinement ordering, congruence) which relate systems whose behaviors are indistinguishable to an external observer.

Axiomatic semantics: Emphasis is put on the axiomatic proof methods to check the correctness of a system against a given specification [31, 34].

Different process algebrae adopt different kind of denotational and axiomatic approaches for specification and verification. For example, CCS has been studied under bisimulation and testing semantics [41], CSP under trace and failure semantics (variants of testing semantics) [21, 49], and ACP under bisimulation and branching bisimulation semantics [17]. In all cases however one needs to establish an operational semantics first. A system often consists of several levels of subsystems. The congruence and refinement relation provided by a process algebra may be used to determine whether these different subsystems conform to one another. These relations are typically substitutive, meaning that related subsystems may be used interchangeably inside a larger system; this provides the facilities for compositional system verification, since low-level subsystems may be checked in isolation from the rest of their high-level subsystem.

Concurrent process algebrae [17, 20, 21, 33, 35, 41, 42, 49] describe a system behavior or a process using eight major operators: event prefix, choice, recursion, parallel composition, hiding, renaming, sequential composition, and interrupt. A prefix or suffix of a process is also a process,

so the domain language must be closed under prefix and suffix. In addition, each operator requires certain prerequisite closure properties of the domain language: closure under union, Kleene star, intersection and shuffle, hiding, renaming, concatenation, prefix are required for choice, recursion, parallel composition, hiding, renaming, sequential composition, and interrupt, respectively. Regular languages have all these required closure properties. Balanced languages (or regular hedge languages) are not closed under prefix or suffix. Deterministic context-free languages are not closed under union, intersection, and concatenation. Context-free languages are not closed under intersection. In all, regular languages are the only domain used for concurrent process algebrae and thus concurrent process algebrae cannot specify non-regular properties. As a result, algebrae are not a dominant technique in software verification.

1.2 Visibly Pushdown Languages

The formal verification arena has been enhanced by the recent introduction of the class of *visibly pushdown languages (VPL)* [10] which lies between balanced languages and deterministic context-free languages. VPL have all the appealing properties that the regular languages enjoy: deterministic acceptors are as expressive as their nondeterministic counterparts; they are closed under union, intersection, complementation, concatenation, Kleene star, prefix, and language homomorphisms; membership, emptiness, language inclusion, and language equivalence are all decidable. VPL are accepted by *visibly pushdown automata (vPDA)* whose stack behaviour is determined by the input. 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 calls and returns can also change the stack content. While reading a call a vPDA must push one symbol on the stack and while reading a return it must pop one symbol (unless the stack is empty). We can model the execution of a recursive module using a VPL by representing the invocation of a module by a call event, the return from a module by a return event, and all the other internal actions by local events. The potential of a VPL-based concurrent process algebra is high, as VPL have all the required closure properties other than hiding and shuffle.

1.3 The Problem

From a conformance testing point of view, vPDA have been mostly studied in terms of logic-based conformance testing (namely, model checking) [2, 8, 10]. vPDA and its natural subclass visibly BPA [51] are also studied, but to a lesser extent in terms of behavioural equivalence such as bisimulation relations. There is to our knowledge no work on process algebra to represent complex, concurrent vPDA systems. One reason behind this is no work established that VPL is closed under shuffle. This being said, concurrency has been studied with *multi-stack visibly pushdown language (MPL)* [53] and *vPDA with two stacks (2-vPDA)* [24]. In [24] authors claim that concurrency is not possible for vPDA although [25] claims that synchronization is possible for these automata. We note that a good concurrent, fully compositional process algebra is the main tool in the virtually unstudied area of vPDA algebraic-based conformance testing.

1.4 The Thesis

Our thesis is that a fully compositional concurrent vPDA-based process algebra is possible. We are thus introducing such an algebra called *Communicating Visibly pushdown Process (CVP)*. We also present the operational semantics and the trace model of CVP.

1.5 Dissertation Summary

In Chapter 2, we present the preliminaries of the dissertation. In Section 3.1 we establish a *labelled transition system (LTS)* semantics for vPDA. LTS are the underlying semantic model for all the process algebrae, so this is one significant step. The underlying LTS of a vPDA-based process algebra is an infinite-state machine. Every state of such an LTS is represented by the combination of a vPDA state and the current stack content associated to that state. We end the aforementioned confusion about vPDA concurrency [24, 25] by showing in Theorem 3.2.2 that VPL is closed under shuffle. We also prove that VPL is closed under hiding (Theorem 3.2.1), so that we achieve all the major prerequisite closure properties for a concurrent, compositional vPDA-based process algebra.

In Chapter 4, we show how the operators of a concurrent process algebra along with a new operator *abstract* can be applied in the VPL setting. We apply our technique on the operators of CSP [20, 21, 33, 35, 49], a finite-state process algebra (a random choice, our formalism works with all the other finite-state process algebrae). We are thus proposing a vPDA-based process algebra called Communicating Visibly pushdown Processes (CVP) as a superset of CSP; when all the input symbols are locals then CVP is equivalent to CSP. In Section 4.1, we introduce the syntax and in Section 4.2, we describe the operational semantics of CVP. In Theorem 4.3.1 we show that CVP is indeed an algebra, being closed under all its operations. In all, we are laying the foundation of VPL-based algebraic specification and verification by introducing a structural operational semantics for CVP.

We then describe the CVP trace model in Chapters 5 and 6. In Chapter 5, we present the trace semantics and in Section 6.1, we define four functions on CVP traces: abstract $\mathfrak{A}$, stack extract $\mathfrak{S}$,module extract $\mathfrak{M}$, and completeness $\mathfrak{C}$. The abstract function hides the traces of the sub-modules from the trace of their parent module, $\mathfrak{S}$ extracts the stack from the trace, $\mathfrak{M}$ extracts the trace of a certain module from a trace, and $\mathfrak{C}$ checks if a trace contains the complete trace of a certain module. These functions work only on the traces in the VPL realm. With the help of these functions we show in Section 6.2 that some very desirable properties for software verification—which cannot be specified in context-free or regular process algebrae—can be specified in CVP. In Section 6.2 we present the trace proof system for CVP, that can be used to verify the properties mentioned in Section 6.2. Chapter 7 concludes the dissertation.

Chapter 2

Preliminaries

2.1 Visibly Pushdown Automata

We denote the empty word and only the empty word by ε.

A visibly pushdown automaton (vPDA) [10] is a tuple $M = (\Phi, \Phi_{in}, \widetilde{\Sigma}, \Gamma, \Omega, \Phi_F)$, where 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 $\perp$, 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 is 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 $\widetilde{\Sigma} = \Sigma_l \uplus \Sigma_c \uplus \Sigma_r$).

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 $a \in \Sigma_c$ 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 [10].

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 word $w = a_1 a_2 \dots a_k$ 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 $\gamma_0 = \bot$, $q_0 \in \Phi_{in}$, $(q_{j-1i},\varepsilon) \xrightarrow{\varepsilon} (q_{ji},\varepsilon) \in \Omega$ 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 words w accepted by M.

2.2 Labelled Transition System

A labelled transition system (LTS) [22] is a tuple (Θ,Σ,Δ,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 $q_0\tau q_{01}\tau\cdots\tau q_{0m_0}a_1q_1\tau q_{11}\tau\cdots\tau q_{1m_1}a_2 q_2\cdots a_kq_k\tau q_{k1}\tau\cdots\tau q_{km_k}$ 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 $a_1a_2\cdots a_k$. 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) [*ctraces*(M)] contains exactly all the traces [complete traces] of all the possible runs [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*

[23]. 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 $p' \sim q'$.

2.3 Communicating Sequential Processes

CSP or Communicating Sequential Processes [20, 21, 33, 35, 49] provides a basis for the study of concurrent computation. A communicating process is regarded as an agent which may interact with its environment (which may itself be regarded as a process) by performing certain instantaneous atomic events drawn from an alphabet Σ. CSP provides a formal language suitable for describing finite-state processes. 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 S and R range over CSP processes, x over Σ, A and B over 2^{Σ}, f over the set $\{f : \Sigma \rightarrow \Sigma : \forall a \in \Sigma\text{: } f^{-1}(a) \text{ is finite} \wedge f(a) = \checkmark \text{ iff } a = \checkmark\}$ of Σ-transformations. The CSP prefix choice $x : A \rightarrow S(x)$ 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 restrictions: 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. LTS is the underlying semantic model of CSP like any other process algebra. The operational semantics of CSP presented on an LTS in Figure 2.1. Thus the whole CSP can be viewed as a single LTS.

The CSP processes $STOP$, $SKIP$ and $a \rightarrow S$ are special instances of the prefix choice construct: $STOP$ is obtained by taking $A = \emptyset$, $SKIP = x : \{\checkmark\} \rightarrow STOP$ and $a \rightarrow S = x : \{a\} \rightarrow S$. The special event ✓ denotes termination. $S \underset{A}{\|} R$ is a special form of $S {}_A\|_B R$; it synchronizes only on those visibly pushdown events appearing in A (S and R interleave for any $a \notin A$). $S \,|||\, R$ is the unrestricted interleaving of S and R and is also a special case of $S {}_A\|_B R$. We denote $A \cup \{\checkmark\}$ by $A^{\checkmark}$.

2.4 Sequences

The set of traces of a process is the set of all the sequences of actions [49] that might possibly be recorded. Such sequences will be described by listing their elements in order between angled brackets. The empty sequence is thus denoted by $\langle\rangle$. If A is a set, then A^* is the set of all finite sequences of elements of A. If $seq1$ and $seq2$ are both sequences, then their concatenation described by $seq1.seq2$ is the sequence of elements in $seq1$ followed by those in $seq2$. The concatenation operation is associative. The notation seq^n describes n copies of the finite sequence seq concatenated together, and so seq^0 is always the empty sequence. If seq is not empty, then it may be written $a.seq'$ where a is the first element of seq, and seq' is the remainder of the sequence. In this case, two functions on seq are defined: $head(seq) = a$ and $tail(seq) = seq'$. For $seq = seq''.b$ we

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

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

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

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

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

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

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

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

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

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

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

Figure 2.1: Operational Semantics of CSP

define $foot(seq) = b$ and $init(seq) = seq''$. The length $|seq|$ of a sequence is the number of elements it contains. The notation $a \in seq$ means that the element a appears in the sequence seq, and $\sigma(seq)$ is the set of all elements that appear in seq.

Various natural relationships between sequences exists: If there is some sequence $seq2$ such that $seq.seq2 = seq1$, then seq is a prefix of $seq1$, written $seq \leq seq1$. Furthermore, $seq \leq_n seq1$ is if $seq \leq seq1$ and their lengths differ by no more than n. If $seq \neq seq1$ then seq is a strict prefix of $seq1$, written $seq < seq1$. The notation $seq \preccurlyeq seq1$ means that seq is a (not necessarily contiguous) subsequence of $seq1$. $seq \restriction A$ is the subsequence of all the elements of seq that are in the set A. Conversely, the notation $seq \setminus A$ is the subsequence of seq whose elements are not in A. If f is a mapping on elements, then $f(seq)$ is the sequence obtained by applying f to each element of seq in turn. *Reverse* or $\mathfrak{R}(seq)$ is a function which reverses a sequence seq: $\mathfrak{R}(seq) = s_n.s_{n-1}.s_{n-2}...s_3.s_2.s_1$ whenever $seq = s_1.s_2.s_3...s_{n-2}.s_{n-1}.s_n$.

2.5 Traces

The concept of traces was briefly introduced in Section 2.2 (and alluded to in Section 2.4); we now present this notion in more detail. Processes interact with their environment through performance of events in their interface. The environment has no direct access to the internal state of the process or to the internal events that it performs. Two processes which are indistinguishable at their interfaces should be equally appropriate for any particular purpose; the way they are implemented cannot have any influence on their respective suitability. There are a number of ways in which interface behavior can be analyzed, but they all concentrate exclusively on the external activity of the process. One important aspect of process behavior concerns the occurrence of events in the right order, and that events do not occur at inappropriate points. The kind of sequence which is acceptable will be given by the requirements of the system. Such requirements will describe constraints on when particular events can occur. The environment cannot know precisely which internal state the process has reached at any particular point, since it has access only to the projection of the execution onto the interface. To analyze process with respect to these requirements, 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 the set of all possible traces of a process P is denoted $traces(P)$.

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. 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 $\exists P' : P \stackrel{tr}{\Rightarrow} P'$.

2.6 Trace Semantics

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. The *trace model* considers processes directly in terms of their traces, and lifts the entire analysis to a more abstract level. All of the operators of the language can be understood at this level: the traces of composite process are dependent only on the traces of its components. This allows a *compositional* semantic model, where all processes are considered only in terms of their sets of traces, and at no stage do the underlying executions need to be considered explicitly.

In the trace model, each process is associated with a set of traces: the set of all possible sequences of events that may be observed during some execution of the process. Processes will be *trace equivalent* when they have exactly the same set of possible traces. This particular form of equality will be denoted $=_T$, and its definition is that $P =_T Q$ *iff* $traces(P) = traces(Q)$. Trace equality gives rise to algebraic laws for individual operators, and also laws concerning the relationships between various operators. These laws allow for the manipulation of process descriptions from one form to another while keeping the associated set of traces unchanged. Many laws are concerned

with general algebraic properties such as associativity and commutativity of operators (which allow components to be composed in any order), idempotence, and the identification of units and zeros for particular operators (which may allow process descriptions to be simplified). Other laws are concerned with the relationships between different operators, which allow for example the expansion of a parallel composition into a prefix choice process.

$STOP$ is a process which cannot do anything:

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

while $SKIP$ can only perform ✓. The only traces $SKIP$ exhibits are the empty trace and the singleton trace containing ✓:

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

One important process to establishing laws in axiomatic model is *RUN* which can do any sequence of events:

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

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

2.7 Specification with Traces

Systems are designed to satisfy particular requirements, and one of the uses of their semantics is to enable them to be judged against a given specification. In the trace model, a specification of a process is given in terms of the traces it may engage in. It will characterize the traces that are acceptable and those that are not. A process meets the specification if all of its executions are acceptable: no matter which choices are taken, any execution of the process is guaranteed not to violate the specification. If $S(tr)$ is a predicate on trace tr, then process P meets (or satisfies) $S(tr)$ if $S(tr)$ holds for every traces tr of P: $P \vdash S(tr) = \forall tr \in traces(P) : S(tr)$. The specification $S(tr)$ is said to be a *property-oriented specification*, since the required property is captured by $S(tr)$

as a restriction on traces. The predicate S may be expressed in any notation, though first order logic and elementary set and sequence notations are generally sufficient.

If a process P fails to meet a specification $S(tr)$, then this must be because it has some (finite) trace for which S fails to hold: there is a point where the performance of a particular event takes the execution of P outside the specification. To meet a trace specification, it is necessary to ensure that no violating events are performed at any stage of an execution. This kind of specification is called a *safety* specification, which requires that nothing 'bad' should ever happen, and it is precisely this kind of property that is expressed as specification on traces.

2.8 Verification with Traces

The compositional nature of the trace semantics allows a compositional proof system to be provided for trace specifications. Specifications of processes may be deduced from the specifications of their components, in a way which reflects the trace semantics of the operators. The proof system is given 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 is due to the nature of $\vdash$ specification, and which therefore hold for all processes. The first is that any process meets the vacuous specification $true(tr)$, which holds for all traces tr:

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

The second 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

$$\frac{\begin{array}{c} P \vdash S(tr) \\ P \vdash T(tr) \end{array}}{P \vdash (S \wedge T)(tr)}.$$

There is only one trace of the process $STOP$: the empty trace. The strongest specification that is met by process $STOP$ is that $tr = \langle\rangle$. This is encapsulated in the rule:

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

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 $STOP$.

The process $SKIP$ does nothing except terminate successfuly. It has only two possible traces, one for the situation before it has terminated successfully, and the other for the situation after. These two traces are $\langle\rangle$ and $\langle\checkmark\rangle$,so the inference rule, which has no antecedents, is the following:

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

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

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

2.9 Previous Work

The *recursive state machines* (RSM) [3, 4, 5] are the first model where researchers discuss the notion of a the pushdown system with entry nodes, exit nodes, and local nodes of a module. RSM can solve major algorithmic problems for model checking including reachability, cycle detection, and language emptiness. RSM defines context-free languages, so concurrency is not possible. Following this work the temporal logic *CARET* [8] comes out with one more constraint which states that one node cannot be used as two or more types: if a node is used as an entry node, then it cannot be used as an exit or local a node. Based on this, CARET partitions all the symbols in three categories: call, return, and local. This classification gives CARET the advantages of pre-post condition

verification, stack analysis, and local properties verification. The same group defines the new class of visibly pushdown language using the same classification of symbols in the underlying alphabet. Due to many appealing properties, from then on the VPL realm has been a very active research area which produced work on congruence for VPL[9], nested words[11], nested trees [7, 6, 26], visibly pushdown games [39], vPDA bisimulation [51], regularity[15], membership problems for VPL [55], minimizing variants of vPDA [28], algorithmic black box conformance testing [36], fixed point [19, 12], first order temporal logic for VPL [2], grammatical representation of VPL[14], specifications for program analysis [27, 44, 48], XML processing [1, 37, 46], transducers [47, 52], and many others [25, 38, 43, 45, 54]. vPDA concurrency has been studied to some degree as mentioned earlier [24, 25]. To the best of our knowledge no work has been done on VPL-based process algebrae.

Chapter 3

Closure Properties of Visibly Pushdown Languages

One of the possible reasons for the absence of a vPDA-based concurrent process algebra is that the known closure properties support such a development only partially. Indeed, suppose that a fully compositional vPDA-based concurrent process algebra (we anticipate a bit and call it CVP) exists, and consider the operation of unrestricted interleaving $|||$ existent in all the finite-state process algebrae. Such an operation is vital, as it models that part of the execution of concurrent processes that do not contain any communication or synchronization; obviously, at some point any two concurrent processes will contain such an execution, so our (for the time being hypothetical) CVP must contain such an interleaving operator. Consider now two processes P_1 and P_2 specified using CVP, whose traces form the languages L_1 and L_2, respectively. It follows that both L_1 and L_2 are visibly pushdown languages, and since the unrestricted interleaving is an operator of CVP (an algebra), the language L of traces of $P_1 ||| P_2$ must also be a visibly pushdown language. However, L is the shuffle of L_1 and L_2. Therefore, a necessary condition for CVP to exist at all is that visibly pushdown languages be closed under shuffle.

Perhaps a less critical but certainly useful operator in a process algebra is hiding. Such an operator is used to hide the internals of a process and expose only its interface to the environment. The same trace argument requires that VPL be closed under hiding for this operator to exists.

We eliminate in this chapter this last stumbling block toward a fully compositional vPDA-based

process algebra, establishing the closure under shuffle and hiding of visibly pushdown languages. In the process, we also establish a semantics for vPDA based on labelled transition systems, which will be a building block in the subsequent development of the mentioned process algebra. These results effectively prove the existence of a vPDA-based process algebra, and also support it by providing a natural semantic mechanism.

3.1 Visibly Pushdown Automata and Labelled Transition Systems

We can define the semantics of vPDA in terms of LTS in a natural way, as follows:

Definition 3.1.1 Given any visibly pushdown automaton $M = (\Phi, \Phi_{in}, \widetilde{\Sigma}, \Gamma, \Omega, \Phi_F)$, the labelled transition system $[\![M]\!]$ is defined as follows: $[\![M]\!] = ((\Phi \cup \{H, I\}) \times \Gamma^*, \widetilde{\Sigma} \cup \{\tau\}, \Delta, (I, \perp))$, where $I, H \notin \Phi$. The transition relation of $[\![M]\!]$ is $\Delta \subseteq ((\Phi \cup \{I\}) \times \Gamma^*) \times (\widetilde{\Sigma} \cup \{\tau\}) \times ((\Phi \cup \{H\}) \times \Gamma^*)$ and is defined as follows: $\Delta = \{((q, \gamma), a, (q', \gamma')) : ((q, \gamma), a, (q', \gamma')) \in \Omega\} \cup \{((I, \perp), \tau, (q, \perp)) : q \in \Phi_{in}\} \cup \{((q, \gamma), \tau, (H, \gamma)) : q \in \Phi_F\} \cup \{((q, \gamma), \tau, (q, \gamma)) : q \notin \Phi_F, \forall a \in \widetilde{\Sigma} \cup \{\tau\} : (q, \gamma) \stackrel{a}{\not\longrightarrow}\}$. ■

A state of $[\![M]\!]$ is labelled with a state of M as well as the stack content associated with that state of M in the given computation. We first include in Δ the transitions corresponding to the transition of the visibly pushdown automaton being modelled. $[\![M]\!]$ should be capable of starting from any state $\Phi_{in} \times \{\perp\}$; to create a unique initial state we introduce a brand new state $(I, \perp)$ and we add to Δ the set of transitions that gets us nondeterministically to one of the initial states of the visibly pushdown automaton being modelled. We invent the final state H that has no outgoing transitions and is reachable from any final state of M via τ transitions. Such a state is useful in the construction of the LTS corresponding to the concatenation of two VPL. Informally, given two LTS with initial (final) states I' and I'' (H' and H''), respectively, the LTS corresponding to the concatenation of the two languages will have I' as initial state, H'' as final state, and all the transitions in the two original LTS plus transitions of form $((H', \gamma), \tau, (I'', \gamma))$. Such a construction will be made formal later. Its correctness will follow from the closure of VPL under concatenation and the availability of

τ transitions that do not change the stack content [10]. Non-final states with no outgoing transitions gain a loop that performs an internal action (so that they cannot participate in a complete trace; the reason will become evident in Theorem 3.1.2).

The following results establish the labelled transition system $[\![M]\!]$ thus constructed as the semantic model of the visibly pushdown automaton M. First, we establish a very strong, bisimilarity-like equivalence:

Theorem 3.1.1 *Let $M = (\Phi, \Phi_{in}, \widetilde{\Sigma}, \Gamma, \Omega, \Phi_F)$ be a visibly pushdown automaton and let $[\![M]\!]$ be the tuple $((\Phi \cup \{H, I\}) \times \Gamma^*, \widetilde{\Sigma} \cup \{\tau\}, \Delta, (I, \bot))$ as constructed in Definition 3.1.1. Then M and $[\![M]\!]$ are* bisimilar*, in the sense that there exists a relation $\sim \subseteq \Phi \times (\Phi \times \Gamma^*)$ such that for every pair $p \sim (q, \gamma)$ and for any $a \in \Sigma \cup \{\varepsilon\}$ and $a' \in \Sigma \cup \{\tau\}$ such that either $a = a'$ or $a = \varepsilon$ and $a' = \tau$:*

1. *Whenever $q \neq I$ and $q' \neq H$, $(p, \alpha) \xrightarrow{a} (p', \alpha') \in \Omega$ implies that $(q, \gamma) \xrightarrow{a'} (q', \gamma')$ such that $\gamma = \alpha\delta$, $\gamma' = \alpha'\delta$ for some $\delta \in \Gamma^*$ and $p' \sim (q', \gamma')$. Conversely,*
2. *Whenever $q \neq I$, $q' \neq H$, $(q, \gamma) \xrightarrow{a'} (q', \gamma')$ implies that either*
   - *(a) $q = q'$, $\gamma = \gamma'$, $a' = \tau$, and $(q, \gamma) \not\xrightarrow{a''} (q'', \gamma'')$ for any $q'' \neq q$, $\gamma \neq \gamma''$, $a'' \in \Sigma \cup \{\tau\}$, or*
   - *(b) $(p, \alpha) \xrightarrow{a} (p', \alpha') \in \Omega$ with $\gamma = \alpha\delta$, $\gamma' = \alpha'\delta$ for some $\delta \in \Gamma^*$ and $p' \sim (q', \gamma')$;*
3. *$p \in \Phi_F$ iff $(q, \gamma) \xrightarrow{\tau} (H, \gamma)$, and $p \in \Phi_{in}$ iff $(I, \bot) \xrightarrow{\tau} (q, \bot)$*

Proof. Items 1 and 3 follow immediately from the definition of $[\![\cdot]\!]$.

We consider the labelled transition system in its unfolded form, i.e., as a tree. The leaves of the tree are either states of form (H, γ), or states (q, γ) that have no outgoing transitions except $(q, \gamma) \xrightarrow{\tau} (q, \gamma)$ (introduced by the definition of $[\![M]\!]$ only for those states q that are not final states of M and for which (q, γ) has no outgoing transitions). The latter are the only states that are not unfolded.

The proof of Item 2 then proceeds by induction over the tree structure of $[\![M]\!]$ as follows: Bisimilarity for leaves is established by Item 3 and Item 2(a) of the definition of $\sim$, respectively. Consider now some non-leaf state (q, γ) with its outgoing transitions $(q, \gamma) \xrightarrow{a'} (q', \gamma')$. For $q \neq I$ and $q' \neq H$, every such a transition comes from a transition in M of form $(q, \alpha) \xrightarrow{a} (q', \alpha')$, with α and α' suitable prefixes of γ and γ', respectively. Such transitions must exist in the original automaton (since it generated the (LTS) transition under scrutiny in the first place), and $q' \sim (q', \gamma')$ by induction hypothesis. ■

In passing, we note that from a language-theoretic point of view a more useful equivalence is in terms of traces. Such an equivalence is readily available:

Theorem 3.1.2 *For any visibly pushdown automaton M it holds that $L(M) =$ ctraces$([\![M]\!])$.*

Proof. Let $M = (\Phi, \Phi_{in}, \widetilde{\Sigma}, \Gamma, \Omega, \Phi_F)$. Consider some $w = a_1 \dots a_k \in L(M)$ and let $(q_0, \bot)(q_{01}, \bot) \cdots (q_{0m_0}, \bot)(q_1, \gamma_1)(q_{11}, \gamma_1) \cdots (q_{1m_1}, \gamma_1)(q_2, \gamma_2) \cdots (q_k, \gamma_k)(q_{k1}, \gamma_k) \cdots (q_{km_k}, \gamma_k)$ be an accepting run of M on w. Then the run $\rho' = (I, \bot)\tau(q_0, \bot)\tau(q_{01}, \bot)\tau \cdots \tau(q_{0m_0}, \bot)a_1 (q_1, \gamma_1)\tau(q_{11}, \gamma_1)\tau \cdots \tau(q_{1m_1}, \gamma_1)a_2(q_2, \gamma_2) \cdots a_k(q_k, \gamma_k)\tau(q_{k1}, \gamma_k)\tau \cdots \tau(q_{km_k}, \gamma_k)\tau(H, \gamma_k)$ exists in $[\![M]\!]$ by definition (indeed, q_0 is an initial state, hence the τ transition from $(I, \bot)$ to $(q_0, \bot)$; similarly, q_{km_k} is a final state, hence the transition from (q_{km_k}, γ_k) to (H, γ_k)). Moreover, ρ' is maximal (since no state (H, γ) has outgoing transitions) and therefore $w \in$ *ctraces*$([\![M]\!])$. Thus, $L(M) \subseteq$ *ctraces*$([\![M]\!])$.

Consider now some $w = a_1 \dots a_k \in$ *ctraces*$([\![M]\!])$. We then have a run $\rho' = (I, \bot)\tau(q_0, \bot)\tau(q_{01}, \bot)\tau \cdots \tau(q_{0m_0}, \bot)a_1(q_1, \gamma_1)\tau(q_{11}, \gamma_1)\tau \cdots \tau(q_{1m_1}, \gamma_1)a_2(q_2, \gamma_2) \cdots a_k(q_k, \gamma_k)\tau(q_{k1}, \gamma_k)\tau \cdots \tau(q_{km_k}, \gamma_k)\tau(H, \gamma_k)$ such that $q_0 \in \Phi_{in}$ and $q_{km_k} \in \Phi_F$. Indeed, the state $(I, \bot)$ has only τ transitions outgoing toward states in $\Phi_{in} \times \{\bot\}$. In addition, exactly all the maximal runs of $[\![M]\!]$ end up in a state (H, γ) (every final state has a τ transition that leads to such a state, and no other state is the terminal state of a maximal run—those non-final states with no outgoing transitions in the original visibly pushdown automaton are given a loop in $[\![M]\!]$ in order to avoid such), so we must end any maximal run at (H, γ).

The preceding state (q_{km_k}, γ_k) is then (a) linked to (H, γ) by a τ transition (only these are available), and (b) has $q_{km_k} \in \Phi_F$ (only such states are linked directly to (H, γ)). Then $(q_0, \bot)(q_{01}, \bot) \cdots (q_{0m_0}, \bot)(q_1, \gamma_1)(q_{11}, \gamma_1) \cdots (q_{1m_1}, \gamma_1)(q_2, \gamma_2) \cdots (q_k, \gamma_k)(q_{k1}, \gamma_k) \cdots (q_{km_k}, \gamma_k)$ is an accepting run of M on w and thus $w \in L(M)$. Therefore *ctraces*$(\llbracket M \rrbracket) \subseteq L(M)$.

In all, $L(M) =$ *ctraces*$(\llbracket M \rrbracket)$, as desired. ■

3.2 Closure Properties of VPL

It is already known that VPL are closed under union, intersection, complementation, renaming, prefix, concatenation, and Kleene star. In addition, we establish in this section the promised closure under hiding and shuffle. We will use in what follows the given vPDAs and also their associated LTS constructed according to Definition 3.1.1. We will then use the relation between the two constructs (vPDA and LTS) as given in Theorem 3.1.1 and Theorem 3.1.2.

3.2.1 Hiding

Given a language L over an alphabet Σ and a set $A \subseteq \Sigma$, the result of hiding A in L is the set $L \setminus A$ that contains exactly all the strings from L but with all the occurrences of symbols in A erased.

Theorem 3.2.1 *VPL are closed under hiding.*

Proof. Consider a VPL L over $\widetilde{\Sigma}$ and any set $A = A_c \uplus A_r \uplus A_l \subseteq \widetilde{\Sigma}$. Let M be a vPDA that accepts L. We show how the symbols in A can be hidden one by one, so that in the end all the symbols from A can be hidden.

Hiding local symbols as well as hiding some call (return) together with all its balanced returns (calls) can be accomplished by simply replacing in M the respective transitions by empty transitions that do not modify the stack (which yields a vPDA). Same goes for hiding unbalanced calls and unbalanced returns.

Consider now that we hide a call c but we do not hide its balanced return r. Then every trace containing c and r in $\llbracket M \rrbracket$ will be transformed from $w_1 c w_2 r w_3$ (with w_2 well-balanced) into $w_1 w_2 r w_3$.

The last call unbalanced in w_1 becomes however balanced with r, and the balance (in w_3) of the other unbalanced calls in w_1 "shift" one symbol to the right (if there is no unbalanced call in w_1 then the return will be unmatched). This shifting is handled by a suitably modified $[\![M]\!]$, in which transitions are added so that the new, shifted balances are allowed: Suppose the original path $w_1cw_2rw_3$ uses the transition $(P,\gamma) \xrightarrow{b} (Q, d\gamma)$ to handle a call from w_1 and $(R, d\delta) \xrightarrow{m} (S, \delta)$ to balance b with a return m from w_3, and suppose that a transition $(P',\gamma') \xrightarrow{b'} (Q', d'\gamma')$ is used to handle the symbol from w_1 that is the first symbol to the left of b in the original path that is either balanced by a return from w_3 or is unbalanced; then, the transition $(P,\gamma) \xrightarrow{b} (Q, d'\gamma)$ is added. Furthermore, one previously balanced return r' in w_3 becomes unbalanced, so whenever the transition $(P, a\perp) \xrightarrow{r'} (Q, \perp)$ is used in the original path, we add to the original definition a transition $(P, \perp) \xrightarrow{r'} (Q, \perp)$. If we perform this procedure for every possible trace $w_1cw_2rw_3$, then we obtain an LTS from which c has been eliminated (that is, hidden). Since we have added only transitions of the proper form, the resulting LTS clearly corresponds to a vPDA.

The introduction of the transition $(P,\gamma) \xrightarrow{b} (Q, d'\gamma)$ causes the introduction of the supplementary transition $(P,\varepsilon) \xrightarrow{b} (Q, d')$ in M. This transition may be (inadvertently) used on other paths than $w_1cw_2rw_3$, thus modifying the original language in an unacceptable manner. To prevent this, we fix a brand new state P'' and we inspect all the paths in $[\![M]\!]$ other than $w_1cw_2rw_3$ for use of a transition $(P,\delta) \xrightarrow{b} (Q, e\delta)$. If such a path is found, we then rename P to P'' on that path (in the preceding transition that will now lead to P'' instead of P and in the current transition which will now start from P'' instead of P). We perform a similar process for the newly introduced transition $(P, a\perp) \xrightarrow{r'} (Q, \perp)$. After all of this is complete, no path other than $w_1cw_2rw_3$ will contain references to P, and thus no other path will be changed by the introduction of these new vPDA transitions.

We perform a similar procedure ("shift" balance, this time to the left) whenever we hide a return r but not its balanced call c; a call c' previously balanced is now unbalanced, but this situation does not need any new transition added to the original LTS, it being handled automatically (since vPDA accept by final state only). ■

Note that this is not an algorithmic procedure (e.g., we can have an infinite number of paths $w_1 c w_2 r w_3$), but does show closure under hiding which serves our purpose.

3.2.2 Shuffle

The shuffle of two languages L_1 and L_2 over an alphabet Σ is defined as $L_1 \,|||\, L_2 = \{w_1 v_1 w_2 v_2 \cdots w_m v_m : w_1 w_2 \cdots w_m \in L_1, v_1 v_2 \cdots v_m \in L_2 \text{ for all } w_i, v_i \in \Sigma^*\}$.

Theorem 3.2.2 *VPL are closed under shuffle.*

Proof. Consider two vPDA $M' = (\Phi', \Phi'_{in}, \widetilde{\Sigma}, \Gamma', \Omega', \Phi'_F)$ and $M'' = (\Phi'', \Phi''_{in}, \widetilde{\Sigma}, \Gamma'', \Omega'', \Phi''_F)$. We will construct the vPDA $M = (\Phi, \Phi_{in}, \widetilde{\Sigma}, \Gamma, \Omega, \Phi_F)$ that accepts the shuffle of $L(M')$ and $L(M'')$. The construction performs an alternative simulation of M' and M'' and is constructed as follows:

We need to keep track of the states of both M' and M'' during any run of M, so we put $\Phi = \Phi' \times \Phi''$. M starts any of its runs from the start of both M' and M'', so we put $\Phi_{in} = \Phi'_{in} \times \Phi''_{in}$. Similarly, at the end of the run M accepts the input iff both M and M' accept their corresponding inputs and thus $\Phi_F = \Phi'_F \times \Phi''_F$. The stack alphabet of M is $\Gamma = \Gamma' \cup \Gamma''$. The transition relation Ω is constructed as follows:

- We can shuffle any symbol with a local in an immediate fashion. If one of the symbols is a local, then it can arbitrarily appear earlier or later than the other symbol in the shuffle. That is, for every
  - pair of symbols x' and x'' such that either $x' \in \Sigma_l$ or $x'' \in \Sigma_l$, and
  - set of rules

$$\begin{aligned} (P', \alpha') \xrightarrow{x'} (Q', \beta') &\in \Omega' \\ (P'', \alpha'') \xrightarrow{x''} (Q'', \beta'')) &\in \Omega'' \end{aligned}$$

  with suitable values for α', β', α'' and β'',

we add the following sets of rules: $\{((P', X), \alpha') \xrightarrow{x'} ((Q', X), \beta') : X \in \{P'', Q''\}\}$ and $\{((X, P''), \alpha'') \xrightarrow{x''} ((X, Q''), \beta'') : X \in \{P', Q'\}\}$.

- Let us shuffle any two call–return pairs in the two languages as if they were alone in the input. In the process the original matchings/balance will change; indeed, if the original matchings/balance are c' and r' in M', and c'' and r'' in M'',"cross-matchings/balance" will be allowed in M between c' and r'' and between c'' and r' (for otherwise a shuffle is not possible). Formally, for every

  – matching call c' and return r' in M',

  – matching call c'' and return r'' in M'', and

  – set of rules

$$\begin{aligned} ((P', \bot) \xrightarrow{c'} (Q', a)), ((R', a) \xrightarrow{r'} (S', \bot)) &\in \Omega' \\ ((P'', \bot) \xrightarrow{c''} (Q'', b)), ((R'', b) \xrightarrow{r''} (S'', \bot)) &\in \Omega'' \end{aligned}$$

  we add the following rules: $\{((P', X), \bot) \xrightarrow{c'} ((Q', X), a) : X \in \{P'', Q''\}\}$, $\{((X, P''), \bot) \xrightarrow{c''} ((X, Q''), b) : X \in \{P', Q'\}\}$, $\{((R', X), \alpha) \xrightarrow{r'} ((S', X), \bot) : X \in \{R'', S''\}, \alpha \in \{a, b\}\}$, and $\{((X, R''), \alpha) \xrightarrow{r''} ((X, S''), \bot) : X \in \{R', S'\}, \alpha \in \{a, b\}\}$.

  In effect, we allow the shuffling of the two pair of symbols in any combination: Whenever M is ready to accept c' it is also ready to accept c''. If one of these two (e.g., c') was already accepted, then M is ready to accept the other symbol (e.g., c''), as well as the matching return of the already accepted input (e.g., r'). Whenever both calls have been accepted, either return is acceptable first. That matching call–return pairs do not exist in isolation but can be mingled with local symbols is taken care of by the previous case.

- We handle an unbalanced call c as follows: Suppose we had a balanced return for c; if this were the case, we would be covered by the previous case. We do not have such a return, but we can however invent one (call it r) in the original vPDA (M' or M'') that contains the

unbalanced call. We then proceed with the construction outlined in the previous case. Once this is done, we hide $\{r\}$ in the resulting language (accepted by M). The call c becomes once more unbalanced. Given Theorem 3.2.1, M continues to be a vPDA.

An unbalanced return is handled similarly: we invent a balance call for it in the original vPDA, we use the previous case to create the vPDA M and then we hide the just invented call.

- Nothing else is included in Ω, for indeed the cases above cover all the possibilities that can appear in a shuffle.

The correctness of the construction follows quite easily from the considerations expressed in each case of the construction (plus Theorem 3.2.1 since the construction uses hiding). ■

Chapter 4

Communicating Visibly pushdown Processes

Starting from the late seventies much attention has been devoted to the research of concurrent finite-state process algebrae such as CSP, CCS, ACP etc. The behavioral semantics of these process algebrae has been modelled by labelled transition systems. Many classes of behavioral equivalence of these process algebrae are now well-established. There are many automatic verification tools for their analysis which incorporate equivalence checking [16, 32]. VPL have all the required closure properties for all the major operators of a concurrent process algebra. As a result, one can pick any concurrent finite-state process algebra and apply its constructions on VPL rather than regular languages to achieve a vPDA-based process algebra. This new process algebra should be more powerful than that finite-state process algebra as VPL is more expressive than regular languages. In this dissertation we choose CSP, a random well-established process algebra to verify our claim.

4.1 Communicating Visibly pushdown Processes

A communicating visibly pushdown (or CVP) process is an agent which interacts with its environment (itself regarded as a process) by performing certain events drawn from a visibly pushdown alphabet $\widetilde{\Sigma} = \Sigma_l \uplus \Sigma_c \uplus \Sigma_r$. The underlying semantics of CVP consists in labelled transition systems where states represent CVP processes. The syntax of CVP will be based on the following

description:

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

where S and R range over CVP processes (to be substantiated later), 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 of CVP and CSP have the similar construction in LTS. $\overline{\cdot}$ is a new operator ("abstract") which hides the sub-modules of a module (further substantiated later).

The discussion in Section 3.1 shows that we can represent an LTS state corresponding to a CVP process as P_γ, where P is the current vPDA state and γ is the current stack content in the vPDA. We refine the CVP syntax as follows:

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

where P, Q, N range over vPDA states and γ and δ represent some (necessarily finite) prefix of the current stack content. In settling the form of γ and δ we note that the transitions of a vPDA (and the associated LTS) depend at most on the top of the stack. Therefore, we sometimes need to mention syntactically the top of the stack only, while other times (before and after a local transition, before a call transition, and after a return transition on a non-empty stack) we do not need to mention any part of the stack. We thus reach the final syntax of CVP: with a and b ranging over $\Gamma \cup \{\varepsilon\}$,

$$\begin{aligned} P_a \quad ::= \quad & x : A \to P(x)_a \mid P_a \Box Q_b \mid P_a \sqcap Q_a \mid N_a \mid P_{aA}\|_B Q_b \mid P_a \setminus A \mid \overline{P_a} \mid f(P_a) \mid \\ & f^{-1}(P_a) \mid P_a; Q_b \mid P_a \triangle Q_b \end{aligned}$$

4.2 The Operational Semantics of CVP

We often use the subscripts l, c and r to denote the sets of local, call and return events, respectively. Any $A \in 2^{\widetilde{\Sigma}}$ will then be the union of three disjoint sets A_l, A_c, A_r. A CVP operation is allowed

$$\frac{}{(x : A \to P(x))_\gamma \xrightarrow{a} P(a)_\gamma} \; [a \in A_l] \qquad \frac{}{(x : A \to P(x))_\gamma \xrightarrow{a} P(a)_{a\gamma}} \; [a \in A_c]$$

$$\frac{}{(x : A \to P(x))_{a\gamma} \xrightarrow{a} P(a)_\gamma} \; [a \in A_r] \qquad \frac{}{(x : A \to P(x))_\perp \xrightarrow{a} P(a)_\perp} \; [a \in A_r]$$

Figure 4.1: Prefix choice

between two CVP processes only when their partitions do not overlap[1] (else the main restriction of VPL over context-free language is violated). For any $a \in \Sigma_l$, $a \in A \cap B$ [$a \in A \setminus B$] is equivalent of $a \in A_l \cap B_l$ [$a \in A_l \setminus B_l$] and so on (for calls and returns). *Matched* call-returns are defined by the specification, while *balanced* call-returns 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. The mapping between the set of calls and the set of stack symbols is always one to one; this helps to extract the stack from a trace and we do not loose generality. In this dissertation, we use the simplest one to one relation that is we will push the call event into the stack as its own corresponding stack symbol. Indeed, the matching calls of a return event are determined at specification time by specifying which stack symbols can be popped by the given return (e.g., if it is specified that $\{a, b\} \subseteq \Sigma_c$ and $c \in \Sigma_r$ which will pop either a or b, then c is the matching return of both a and b). Due to the existence of the call and return events in CVP, we can model the CVP processes as recursive modules. The process between a call and its corresponding return can represent as a (sub-) module. So the top level process is model of the main module. A call event is used for calling a module, a return event is used for returning from a module, and a local event is used for other actions. In CVP one call event cannot be used to call two different modules but more than one call event can call the same module; a similar restriction holds for return events. The stack grows to the left, hence the bottom-of-stack $\perp$ gets the rightmost place.

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

Figure 4.2: Internal transition (a), internal choice (b)

4.2.1 Prefix Choice

The semantics of prefix choice is shown in Figure 4.1. The event prefix can be introduced by eight kinds of syntactic rules: $P = a \rightarrow P'$, $P = b \rightarrow P'_b$, $P_c = c \rightarrow P'$, $P_\perp = d \rightarrow P'_\perp$, $P_e = STOP$, $P_e = SKIP$, $P = STOP$, and $P = SKIP$. From the semantics of prefix choice we then recognize that a is local, b is a call, c is a balanced return, and d is an unbalanced return. $P_e = STOP$ [$P_e = SKIP$] requires that the process enter the $STOP$ [$SKIP$] state only when the vPDA state is P and the top of the stack is e. On the other hand, $P = STOP$ [$P = SKIP$] does not impose any constraints on the top of the stack.

In general, we can specify any system with as well as without an explicit partitioning of its events. However, if we do not provide an explicit partition, then we can write a process as a sequence of events (like in CSP) only when all the events are locals. On the other hand, we can write any finite process (including processes with calls and returns) as a sequence (desirable in a large system), provided that we specify a partition on its events. For example, $P_\perp$ is a process without an explicit partition: $P = a \rightarrow P_a$, $P = b \rightarrow P1$, $P1 = e \rightarrow P2_e$, $P2 = d \rightarrow P3$, $P3_e = f \rightarrow P4$, $P4_a = c \rightarrow P4$, $P4_\perp = STOP$. The process can be written with an explicit partition as $A_l = \{b, d\}$, $A_c = \{a, e\}$, $A_r = \{c, f\}$; $P = a \rightarrow P_a$, $P = b \rightarrow e \rightarrow d \rightarrow P3_e$, $P3_e = c \rightarrow P4$, $P4_a = c \rightarrow P4$, $P4_\perp = STOP$ or $P = a_c \rightarrow P_a$, $P = b \rightarrow e_c \rightarrow d \rightarrow P3_e$, $P3_e = c_r \rightarrow P4$, $P4_a = c_r \rightarrow P4$, $P4_\perp = STOP$.

4.2.2 Internal Event

A CVP process can perform the internal event τ (not noticeable to the environment), which is able to change the current vPDA state but is unable to affect the vPDA stack. The behaviour of the τ

[1]Meaning that $\Sigma'_x \cap \Sigma''_y = \emptyset$ for all $x \neq y$ for two partitions $\widetilde{\Sigma}' = \{\Sigma'_c, \Sigma'_r, \Sigma'_l\}$ and $\widetilde{\Sigma}'' = \{\Sigma''_c, \Sigma''_r, \Sigma''_l\}$.

$$\frac{P_\gamma \xrightarrow{\tau} P'_\gamma}{\begin{array}{c}P_\gamma \Box Q_\delta \xrightarrow{\tau} P'_\gamma \Box Q_\delta \\ Q_\delta \Box P_\gamma \xrightarrow{\tau} Q_\delta \Box P'_\gamma\end{array}} \qquad \frac{P_\gamma \xrightarrow{a} P'_\gamma}{\begin{array}{c}P_\gamma \Box Q_\delta \xrightarrow{a} P'_\gamma \\ Q_\delta \Box P_\gamma \xrightarrow{a} P'_\gamma\end{array}} \qquad \frac{P_\gamma \xrightarrow{a} P'_{a\gamma}}{\begin{array}{c}P_\gamma \Box Q_\delta \xrightarrow{a} P'_{a\gamma} \\ Q_\delta \Box P_\gamma \xrightarrow{a} P'_{a\gamma}\end{array}}$$

$$\frac{P_{a\gamma} \xrightarrow{a} P'_\gamma}{\begin{array}{c}P_{a\gamma} \Box Q_{a\delta} \xrightarrow{a} P'_\gamma \\ Q_{a\delta} \Box P_{a\gamma} \xrightarrow{a} P'_\gamma\end{array}} \qquad \frac{P_\perp \xrightarrow{a} P'_\perp}{\begin{array}{c}P_\perp \Box Q_\perp \xrightarrow{a} P'_\perp \\ Q_\perp \Box P_\perp \xrightarrow{a} P'_\perp\end{array}} \qquad \frac{P_\gamma \xrightarrow{\checkmark} P'_\gamma}{\begin{array}{c}P_\gamma \Box Q_\delta \xrightarrow{\checkmark} P'_\gamma \\ Q_\delta \Box P_\gamma \xrightarrow{\checkmark} P'_\gamma\end{array}}$$

Figure 4.3: External choice

$$\frac{P_\gamma \xrightarrow{\tau} P'_\gamma}{N_\gamma \xrightarrow{\tau} P'_\gamma}\ [N = P] \qquad \frac{P_\gamma \xrightarrow{a} P'_\gamma}{N_\gamma \xrightarrow{a} P'_\gamma}\ [N = P]$$

$$\frac{P_\gamma \xrightarrow{a} P'_{a\gamma}}{N_\gamma \xrightarrow{a} P'_{a\gamma}}\ [N = P] \qquad \frac{P_{a\gamma} \xrightarrow{a} P'_\gamma}{N_{a\gamma} \xrightarrow{a} P'_\gamma}\ [N = P]$$

$$\frac{P_\perp \xrightarrow{a} P'_\perp}{N_\perp \xrightarrow{a} P'_\perp}\ [N = P] \qquad \frac{P_\gamma \xrightarrow{\checkmark} P'_\gamma}{N_\gamma \xrightarrow{\checkmark} P'_\gamma}\ [N = P]$$

Figure 4.4: Recursion

transition is described in Figure 4.2(a).

4.2.3 Choice

The semantics of internal and external choice are shown in Figures 4.2(b) and 4.3. Choice cannot change the matched call-returns; the stack of the composite process is similar to the stacks of component processes. A process that chooses (once!) between '$]$' and '$\rangle$' as balanced return for '$[$' can be defined as follows: $P = [\rightarrow P_{[}, P = P1 \Box P2, P1_{[} =] \rightarrow P1, P2_{[} =\rangle \rightarrow P2, P1_\perp = STOP, P2_\perp = STOP$. Note that '$]$' and '$\rangle$' are both matching returns of '$[$' but only one is used as balanced return, depending on the environment.

4.2.4 Recursion

A CSP recursive process creates a loop among LTS states while CVP recursive processes create loops among vPDA states only. During each recursive loop a CVP process will change the stack

by the same amount, as in each loop it will visit the same vPDA states and will execute the same visible actions. If the amount of change is zero, then the recursive process can be represented by a finite state machine; it also creates a loop among LTS states. The condition for a CVP process P_γ to be called recursive is that the process definition contains the vPDA state P.

The semantics of recursion is shown in Figure 4.4. Like CSP, CVP supports right- and left-embedding recursion, but it also supports self-embedding recursion such as balanced brackets: $P = (\rightarrow P_(, P_(=) \rightarrow P, P_\perp = STOP$. $P_\perp$ can produce an infinite number of LTS states and infinitely many possible traces (see Figure 4.5) although we only have one vPDA state P. Consider now the process $Q = (\rightarrow Q1_(, Q1_(=) \rightarrow Q, Q_\perp = STOP$. It may have infinitely long traces and can be written as follows: $Q = (\rightarrow) \rightarrow Q, Q_\perp = STOP$. One can argue that the events "(" and ")" are behaving like locals in $Q_\perp$, as $Q_\perp$ can be represented by a finite state machine. However, in $P_\perp$ above the event "(" *must* be a call and the event ")" *must* be a return. Any CVP composition between $Q_\perp$ and $P_\perp$ is possible only if $(\in \Sigma_c$ and $) \in \Sigma_r$, for otherwise the partitions of $P_\perp$ and $Q_\perp$ will not coincide. It is therefore recommended that the partitioning be made in a process-dependent way, and not in order to simplify the process definition.

A recursive process can produce an unbounded stack and cause the system to crash. Stack height of a non-recursive process is bounded by the total number of the call events occurring in the process definition. A right-embedding (or left-embedding) recursion cannot produce an unbounded stack if the number of returns is greater than or equal to the number of calls in the process definition (e.g. in $Q = (\rightarrow Q1_(, Q1_(=) \rightarrow Q, Q_\perp = STOP$); otherwise (e.g. in $Q = (\rightarrow Q, Q_\perp = STOP$) it can produce an unbounded stack. A self-embedding recursive process (e.g. balanced brackets) features an iteratively increasing part followed by an iteratively decreasing part. The increasing part may run ad infinitum and produce an unbounded stack. A stack inspection interrupt process (interrupt will be defined later) should therefore be used with left- or right-embedding recursive processes that have more call event than return event occurrences in their process definition, as well as with any self-embedding recursive process.

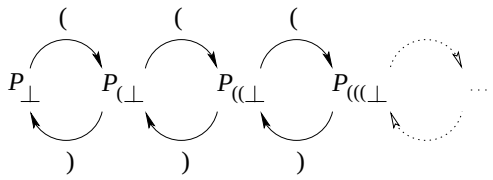

Figure 4.5: Balanced brackets

4.2.5 Parallel Composition

Figure 4.6 shows the semantics of parallel composition. Any common event of the component processes must synchronize during execution. Synchronization is symmetric and instantaneous, so the composite process performs only one event and pushes [pops] only one stack symbol onto [off] the composite stack. The form of the semantics shown in Figure 4.7(a) explains this in detail for synchronized call and return transitions (using explicitly the composite stack). The component processes perform independently the events which are not common between them. Thus every interleaved call [return] event pushes [pops] one stack symbol to the composite stack. Figure 4.7(b) shows the detailed semantics (using the composite stack explicitly) for unsynchronized call and return transitions.

In parallel composition if one process performs an unsynchronized call and then the other wants to perform an unsynchronized return, the second process will pop its own stack and then the composite process will pop the top of its stack (pushed by the first process). Hence in the composite process balanced call-returns depend not only on its components, but also on the sequence of execution of the components. Unsynchronized execution can change the balanced call-returns in the composite process. To illustrate this, consider $P = [\rightarrow P1_{[}, P1 = \langle \rightarrow P2_{\langle}, P2_{\langle} = \rangle \rightarrow P3, P3_{[} =] \rightarrow STOP$ and $Q = [\rightarrow Q1_{[}, Q1 = \lfloor \rightarrow Q2_{\lfloor}, Q2_{\lfloor} = \rfloor \rightarrow Q3, Q3_{[} =] \rightarrow STOP$. The process $P_{\perp\{[,\langle,\rangle,]\}}\|_{\{[,\lfloor,\rfloor,]\}}Q_{\perp}$ is shown in Figure 4.8. In $P_{\perp\{[,\langle,\rangle,]\}}\|_{\{[,\lfloor,\rfloor,]\}}Q_{\perp}$ the matching return of '$\langle$' and '$\lfloor$' are '$\rangle$' and '$\rfloor$'. In the runs A–B–C–E–I–M–$STOP$ and A–B–D–G–K–O–$STOP$, '$\rangle$' is the balanced return of '$\lfloor$' for $P_{\perp\{[,\langle,\rangle,]\}}\|_{\{[,\lfloor,\rfloor,]\}}Q_{\perp}$, even if '$\rangle$' is the sole matched return of '$\langle$' in P. Parallel composition of balanced processes can also create unbalanced process.

$$\frac{P_\gamma \xrightarrow{a} P'_\gamma \qquad Q_\delta \xrightarrow{a} Q'_\delta}{P_{\gamma A}\|_B Q_\delta \xrightarrow{a} P'_{\gamma A}\|_B Q'_\delta}\ [a \in A \cap B] \qquad \frac{P_\gamma \xrightarrow{a} P'_{a\gamma} \qquad Q_\delta \xrightarrow{a} Q'_{a\delta}}{P_{\gamma A}\|_B Q_\delta \xrightarrow{a} P'_{a\gamma A}\|_B Q'_{a\delta}}\ [a \in A \cap B]$$

$$\frac{P_{a\gamma} \xrightarrow{a} P'_\gamma \qquad Q_{b\delta} \xrightarrow{a} Q'_\delta}{P_{a\gamma A}\|_B Q_{b\delta} \xrightarrow{a} P'_{\gamma A}\|_B Q'_\delta}\ [a \in A \cap B] \qquad \frac{P_\perp \xrightarrow{a} P'_\perp \qquad Q_\perp \xrightarrow{a} Q'_\perp}{P_{\perp A}\|_B Q_\perp \xrightarrow{a} P'_{\perp A}\|_B Q'_\perp}\ [a \in A \cap B]$$

$$\frac{P_{a\gamma} \xrightarrow{a} P'_\gamma \qquad Q_\perp \xrightarrow{a} Q'_\perp}{\begin{array}{l} P_{a\gamma A}\|_B Q_\perp \xrightarrow{a} P'_{\gamma A}\|_B Q'_\perp \\ Q_{\perp B}\|_A P_{a\gamma} \xrightarrow{a} Q_{\perp B}\|_A P'_\gamma \end{array}}\ [a \in A \cap B] \qquad \frac{P_\gamma \xrightarrow{a} P'_\gamma}{\begin{array}{l} P_{\gamma A}\|_B Q_\delta \xrightarrow{a} P'_{\gamma A}\|_B Q_\delta \\ Q_{\delta B}\|_A P_\gamma \xrightarrow{a} Q_{\delta B}\|_A P'_\gamma \end{array}}\ [a \in A \setminus B]$$

$$\frac{P_\gamma \xrightarrow{a} P'_{a\gamma}}{\begin{array}{l} P_{\gamma A}\|_B Q_\delta \xrightarrow{a} P'_{a\gamma A}\|_B Q_\delta \\ Q_{\delta B}\|_A P_\gamma \xrightarrow{a} Q_{\delta B}\|_A P'_{a\gamma} \end{array}}\ [a \in A \setminus B] \qquad \frac{P_{a\gamma} \xrightarrow{a} P'_\gamma}{\begin{array}{l} P_{a\gamma A}\|_B Q_\delta \xrightarrow{a} P'_{\gamma A}\|_B Q_\delta \\ Q_{\delta B}\|_A P_{a\gamma} \xrightarrow{a} Q_{\delta B}\|_A P'_\gamma \end{array}}\ [a \in A \setminus B]$$

$$\frac{P_\perp \xrightarrow{a} P'_\perp}{\begin{array}{l} P_{\perp A}\|_B Q_\delta \xrightarrow{a} P'_{\perp A}\|_B Q_\delta \\ Q_{\delta B}\|_A P_\perp \xrightarrow{a} Q_{\delta B}\|_A P'_\perp \end{array}}\ [a \in A \setminus B] \qquad \frac{P_\gamma \xrightarrow{\tau} P'_\gamma}{\begin{array}{l} P_{\gamma A}\|_B Q_\delta \xrightarrow{\tau} P'_{\gamma A}\|_B Q_\delta \\ Q_{\delta B}\|_A P_\gamma \xrightarrow{\tau} Q_{\delta B}\|_A P'_\gamma \end{array}}$$

$$\frac{P_\gamma \xrightarrow{\checkmark} P'_\gamma \qquad Q_\delta \xrightarrow{\checkmark} Q'_\delta}{P_{\gamma A}\|_B Q_\delta \xrightarrow{\checkmark} P'_{\gamma A}\|_B Q'_\delta}$$

Figure 4.6: Alphabetized parallel

4.2.6 Hiding

Figure 4.9 shows the semantics of hiding. The stack of the process does not change when a hidden call or return is performed; part of Figure 4.9 could be written in the possibly clearer but more elaborate form from Figure 4.10. If we hide ‘)’ in the process $P_\perp$ of balanced brackets used earlier, then the traces of the resulting process will form the language $(^*$. Hiding transforms here a balanced process into an unbalanced one. Hiding can change unbalanced [balanced] processes into balanced [unbalanced] ones; it can also change the matched call-returns.

4.2.7 Abstract

The operator introduced over any other process algebra is abstract, which hides all the sub-modules of a module. This operator is motivated by the abstract path in CARET and NWTL [2, 8]. One

$$\frac{P_\gamma \xrightarrow{a} P'_{a\gamma} \qquad Q_\delta \xrightarrow{a} Q'_{a\delta}}{(P_{\gamma A} \|_B Q_\delta)_\phi \xrightarrow{a} (P'_{a\gamma A} \|_B Q'_{a\delta})_{a\phi}} \begin{bmatrix} a \in \\ A \cap B \end{bmatrix} \qquad \frac{P_{a\gamma} \xrightarrow{a} P'_\gamma \qquad Q_{b\delta} \xrightarrow{a} Q'_\delta}{\begin{array}{l}(P_{a\gamma A} \|_B Q_{b\delta})_{c\phi} \xrightarrow{a} (P'_{\gamma A} \|_B Q'_\delta)_\phi \\ (P_{a\gamma A} \|_B Q_{b\delta})_\perp \xrightarrow{a} (P'_{\gamma A} \|_B Q'_\delta)_\perp\end{array}} \begin{bmatrix} a \in \\ A \cap B \end{bmatrix}$$

(a)

$$\frac{P_\gamma \xrightarrow{a} P'_{a\gamma}}{\begin{array}{l}(P_{\gamma A} \|_B Q_\delta)_\phi \xrightarrow{a} (P'_{a\gamma A} \|_B Q_\delta)_{a\phi} \\ (Q_{\delta B} \|_A P_\gamma)_\phi \xrightarrow{a} (Q_{\delta B} \|_A P'_{a\gamma})_{a\phi}\end{array}} \begin{bmatrix} a \in \\ A \setminus B \end{bmatrix} \qquad \frac{P_{a\gamma} \xrightarrow{a} P'_\gamma}{\begin{array}{l}(P_{a\gamma A} \|_B Q_\delta)_{c\phi} \xrightarrow{a} (P'_{\gamma A} \|_B Q_\delta)_\phi \\ (Q_{\delta B} \|_A P_{a\gamma})_{c\phi} \xrightarrow{a} (Q_{\delta B} \|_A P'_\gamma)_\phi \\ (P_{a\gamma A} \|_B Q_\delta)_\perp \xrightarrow{a} (P'_{\gamma A} \|_B Q_\delta)_\perp \\ (Q_{\delta B} \|_A P_{a\gamma})_\perp \xrightarrow{a} (Q_{\delta B} \|_A P'_\gamma)_\perp\end{array}} \begin{bmatrix} a \in \\ A \setminus B \end{bmatrix}$$

(b)

Figure 4.7: Alternate semantics of parallel composition

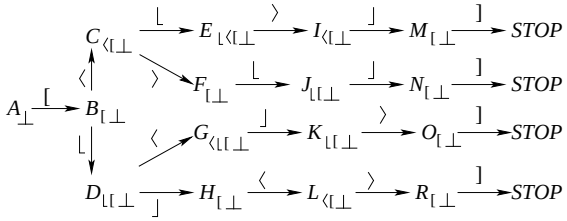

Figure 4.8: Example of parallel composition

can now hide the sub-modules from the environment. This cannot be accomplished using the hide operator if an event is present both in the module and its sub-module. Abstract produces the local trace of a module, so that one can specify internal properties of a recursive module. The semantics of abstract is presented in Figure 4.11. During the execution of a call, abstract pushes the corresponding stack symbol with two special markers: $\tilde{\cdot}$ for the internal call of the main module and $\bar{\cdot}$ for the internal call of a sub-module. If the top of the stack contains any special marker then every local event will be hidden; calls and returns are pushed to/popped off the stack but are otherwise hidden as well (except for top-level calls and returns in the module). If a return occurs when the top of the stack is not marked, the process will get out of abstract.

Let B (with call b and return f), and C (with call d and return e) be two modules. The top-level process P calls B and B calls C: $P = a \rightarrow Q$, $Q = b \rightarrow R_b$, $R = c \rightarrow S$, $S = d \rightarrow T_d$,

$$\frac{P_\gamma \xrightarrow{t} P'_\gamma}{P_\gamma \setminus A \xrightarrow{t} P'_\gamma \setminus A} \; [t \in \{\tau, \checkmark\}] \qquad \frac{P_\gamma \xrightarrow{a} P'_\gamma}{P_\gamma \setminus A \xrightarrow{\tau} P'_\gamma \setminus A} \; [a \in A]$$

$$\frac{P_\gamma \xrightarrow{a} P'_{a\gamma}}{P_\gamma \setminus A \xrightarrow{\tau} P'_{a\gamma} \setminus A} \; [a \in A] \qquad \frac{P_{a\gamma} \xrightarrow{a} P'_\gamma}{P_{a\gamma} \setminus A \xrightarrow{\tau} P'_\gamma \setminus A} \; [a \in A]$$

$$\frac{P_\perp \xrightarrow{a} P'_\perp}{P_\perp \setminus A \xrightarrow{\tau} P'_\perp \setminus A} \; [a \in A] \qquad \frac{P_\gamma \xrightarrow{a} P'_\gamma}{P_\gamma \setminus A \xrightarrow{a} P'_\gamma \setminus A} \; [a \notin A]$$

$$\frac{P_\gamma \xrightarrow{a} P'_{a\gamma}}{P_\gamma \setminus A \xrightarrow{a} P'_{a\gamma} \setminus A} \; [a \notin A] \qquad \frac{P_{a\gamma} \xrightarrow{a} P'_\gamma}{P_{a\gamma} \setminus A \xrightarrow{a} P'_\gamma \setminus A} \; [a \notin A]$$

$$\frac{P_\perp \xrightarrow{a} P'_\perp}{P_\perp \setminus A \xrightarrow{a} P'_\perp \setminus A} \; [a \notin A]$$

Figure 4.9: Hiding

$$\frac{P_\gamma \xrightarrow{a} P'_{a\gamma}}{(P_\gamma \setminus A)_\phi \xrightarrow{\tau} (P'_{a\gamma} \setminus A)_\phi} \; [a \in A] \qquad \frac{P_{a\gamma} \xrightarrow{a} P'_\gamma}{(P_{a\gamma} \setminus A)_\phi \xrightarrow{\tau} (P'_\gamma \setminus A)_\phi} \; [a \in A]$$

Figure 4.10: Altenrate semantics of hiding

$T = c \to U$, $U_d = e \to V$, $V_b = f \to W$, and $W = c \to STOP$. We can hide the sub-modules of B by using abstract: $P = a \to Q$, $Q = b \to \overline{R_b}$, $R = c \to S$, $S = d \to T_d$, $T = c \to U$, $U_d = e \to V$, $V_b = f \to W$, and $W = c \to STOP$. We actually hide sub-module C in this particular example.

4.2.8 Renaming

The semantics of forward and backward renaming is depicted in Figures 4.12 and 4.13. Renaming functions cannot change the VPL partition. Renaming can modify the matched call-returns of a process but cannot change a balanced [unbalanced] process into an unbalanced [balanced] one. There might be no "reverse" renaming that retrieves the original process or set of matched call-returns: If we apply a renaming $f(\langle) =]$ on our previous example (illustrating choice), we get a

$$\frac{P_{b\gamma} \xrightarrow{a} P'_{ab\gamma}}{\begin{array}{l}\overline{P_{b\gamma}} \xrightarrow{a} \overline{P'_{\tilde{a}b\gamma}} \\ \overline{P_{\tilde{b}\gamma}} \xrightarrow{\tau} \overline{P'_{\overline{\tilde{a}}b\gamma}} \\ \overline{P_{\overline{b}\gamma}} \xrightarrow{\tau} \overline{P'_{\overline{a}\overline{b}\gamma}}\end{array}} \qquad \frac{P_{a\gamma} \xrightarrow{a} P'_{a\gamma}}{\begin{array}{l}\overline{P_{a\gamma}} \xrightarrow{a} \overline{P'_{a\gamma}} \\ \overline{P_{\tilde{a}\gamma}} \xrightarrow{\tau} \overline{P'_{\tilde{a}\gamma}} \\ \overline{P_{\overline{a}\gamma}} \xrightarrow{\tau} \overline{P'_{\overline{a}\gamma}}\end{array}} \qquad \frac{P_{a\gamma} \xrightarrow{a} P'_{\gamma}}{\begin{array}{l}\overline{P_{a\gamma}} \xrightarrow{a} \overline{P'_{\gamma}} \\ \overline{P_{\tilde{a}\gamma}} \xrightarrow{a} \overline{P'_{\gamma}} \\ \overline{P_{\overline{a}\gamma}} \xrightarrow{\tau} \overline{P'_{\gamma}}\end{array}}$$

$$\frac{P_{\gamma} \xrightarrow{\tau} P'_{\gamma}}{\overline{P_{\gamma}} \xrightarrow{\tau} \overline{P'_{\gamma}}} \qquad \frac{P_{\gamma} \xrightarrow{\checkmark} P'_{\gamma}}{\overline{P_{\gamma}} \xrightarrow{\checkmark} P'_{\gamma}} \qquad \frac{P_{\perp} \xrightarrow{a} P'_{\perp}}{\overline{P_{\perp}} \xrightarrow{a} P'_{\perp}} \ [a \in A_r]$$

Figure 4.11: Abstract

$$\frac{P_{\gamma} \xrightarrow{\tau} P'_{\gamma}}{f(P)_{\gamma} \xrightarrow{\tau} f(P')_{\gamma}} \qquad \frac{P_{\gamma} \xrightarrow{a} P'_{\gamma}}{f(P)_{\gamma} \xrightarrow{f(a)} f(P')_{\gamma}} \qquad \frac{P_{\gamma} \xrightarrow{a} P'_{a\gamma}}{f(P)_{\gamma} \xrightarrow{f(a)} f(P')_{f(a)\gamma}}$$

$$\frac{P_{a\gamma} \xrightarrow{a} P'_{\gamma}}{f(P)_{f(a)\gamma} \xrightarrow{f(a)} f(P')_{\gamma}} \qquad \frac{P_{\perp} \xrightarrow{a} P'_{\perp}}{f(P)_{\perp} \xrightarrow{f(a)} f(P')_{\perp}} \qquad \frac{P_{\gamma} \xrightarrow{\checkmark} P'_{\gamma}}{f(P)_{\gamma} \xrightarrow{\checkmark} f(P')_{\gamma}}$$

Figure 4.12: Forward renaming

process whose traces define the language $[^n]^n$; no renaming can give back the original.

4.2.9 Sequential Composition and Interrupt

Figures 4.14 and 4.15 show the semantics of sequential composition and interrupt. These operators can change the matched call-returns.

$$\frac{P_{\gamma} \xrightarrow{\tau} P'_{\gamma}}{f^{-1}(P)_{\gamma} \xrightarrow{\tau} f^{-1}(P')_{\gamma}} \qquad \frac{P_{\gamma} \xrightarrow{f(a)} P'_{\gamma}}{f^{-1}(P)_{\gamma} \xrightarrow{a} f^{-1}(P')_{\gamma}} \qquad \frac{P_{\gamma} \xrightarrow{f(a)} P'_{f(a)\gamma}}{f^{-1}(P)_{\gamma} \xrightarrow{a} f^{-1}(P')_{a\gamma}}$$

$$\frac{P_{f(a)\gamma} \xrightarrow{f(a)} P'_{\gamma}}{f^{-1}(P)_{a\gamma} \xrightarrow{a} f^{-1}(P')_{\gamma}} \qquad \frac{P_{\perp} \xrightarrow{f(a)} P'_{\perp}}{f^{-1}(P)_{\perp} \xrightarrow{a} f^{-1}(P')_{\perp}} \qquad \frac{P_{\gamma} \xrightarrow{\checkmark} P'_{\gamma}}{f^{-1}(P)_{\gamma} \xrightarrow{\checkmark} f^{-1}(P')_{\gamma}}$$

Figure 4.13: Backward renaming

$$\frac{P_\gamma \xrightarrow{\tau} P'_\gamma}{P_\gamma; Q_\delta \xrightarrow{\tau} P'_\gamma; Q_\delta} \qquad \frac{P_\gamma \xrightarrow{a} P'_\gamma}{P_\gamma; Q_\delta \xrightarrow{a} P'_\gamma; Q_\delta} \qquad \frac{P_\gamma \xrightarrow{a} P'_{a\gamma}}{P_\gamma; Q_\delta \xrightarrow{a} P'_{a\gamma}; Q_\delta}$$

$$\frac{P_{a\gamma} \xrightarrow{a} P'_\gamma}{P_{a\gamma}; Q_\delta \xrightarrow{a} P'_\gamma; Q_\delta} \qquad \frac{P_\perp \xrightarrow{a} P'_\perp}{P_\perp; Q_\delta \xrightarrow{a} P'_\perp; Q_\delta} \qquad \frac{P_\gamma \xrightarrow{\checkmark} P'_\gamma}{P_\gamma; Q_\delta \xrightarrow{\tau} Q_\delta}$$

Figure 4.14: Sequential composition

$$\frac{P_\gamma \xrightarrow{\tau} P'_\gamma}{P_\gamma \triangle Q_\delta \xrightarrow{\tau} P'_\gamma \triangle Q_\delta} \qquad \frac{P_\gamma \xrightarrow{a} P'_\gamma}{P_\gamma \triangle Q_\delta \xrightarrow{a} P'_\gamma \triangle Q_\delta} \qquad \frac{P_\gamma \xrightarrow{a} P'_{a\gamma}}{P_\gamma \triangle Q_\delta \xrightarrow{a} P'_{a\gamma} \triangle Q_\delta}$$

$$\frac{P_{a\gamma} \xrightarrow{a} P'_\gamma}{P_{a\gamma} \triangle Q_\delta \xrightarrow{a} P'_\gamma \triangle Q_\delta} \qquad \frac{P_\perp \xrightarrow{a} P'_\perp}{P_\perp \triangle Q_\delta \xrightarrow{a} P'_\perp \triangle Q_\delta} \qquad \frac{P_\gamma \xrightarrow{\checkmark} P'_\gamma}{P_\gamma \triangle Q_\delta \xrightarrow{\checkmark} P'_\gamma}$$

$$\frac{Q_\delta \xrightarrow{\tau} Q'_\delta}{P_\gamma \triangle Q_\delta \xrightarrow{\tau} P_\gamma \triangle Q'_\delta} \qquad \frac{Q_\delta \xrightarrow{a} Q'_{a\delta}}{P_\gamma \triangle Q_\delta \xrightarrow{a} Q'_{a\delta}} \qquad \frac{Q_\delta \xrightarrow{a} Q'_\delta}{P_\gamma \triangle Q_\delta \xrightarrow{a} Q'_\delta}$$

$$\frac{Q_{a\delta} \xrightarrow{a} Q'_\delta}{P_\gamma \triangle Q_{a\delta} \xrightarrow{a} Q'_\delta} \qquad \frac{Q_\perp \xrightarrow{a} Q'_\perp}{P_\gamma \triangle Q_\perp \xrightarrow{a} Q'_\perp}$$

Figure 4.15: Interrupt

4.3 CVP Is a Process Algebra

Theorem 4.3.1 *CVP is an algebra; that is, CVP is closed under all its operators. The underlying semantics of any CVP process is a vPDA (or an equivalent LTS).*

Proof. We proceed by structural induction. $STOP$, $SKIP$ are obviously CVP processes. It is also easy to see that CVP is closed under: prefix choice (which just follows the definition of a transition in the associated vPDA), external choice (which is a prefix choice with more than one alternative; many transitions out of one state are clearly allowed), internal choice (we connect two LTS to a common start state via τ transitions that does not change the stack), recursion (which generates a loop from some vPDA state P back into P; this does not introduce infinite vPDA states, and manipulates the stack according to the vPDA semantics introduced by the other transitions),

renaming (unchanged VPL partition), and abstract (we replace whole portions of the LTS with τ transitions).

Hiding is straightforward except when we hide a call c but we do not hide its balanced return r (or the other way around). Then every path containing c and r will be transformed from $w_1cw_2rw_3$ (with w_2 balanced) into $w_1w_2rw_3$. The last call unbalanced in w_1 becomes however balanced with r, and the matching (in w_3) of the other unbalanced calls in w_1 "shift" one symbol to the right. This shifting is handled by adding rules so that the new, shifted matchings are allowed: Suppose the original path $w_1cw_2rw_3$ uses the rules $P = b \to Q_d$ to handle a call from w_1 and $R_d = m \to S$ to match b with a return m from w_3, and suppose that a rule $P' = b' \to Q'_{\mathfrak{d}'}$ is used to handle the symbol from w_1 that is the first symbols to the left of b in the original path that is either balanced by a return from w_3 or is unbalanced; then, the rule $P = b \to Q_{\mathfrak{d}'}$ is added. One previously balanced return r' in w_3 becomes unbalanced, so whenever the rule $P_a = r' \to Q$ is used in the original path, we add a rule $P_\perp = r' \to Q$ (we change the original LTS, but it is easier to describe the process in terms of adding rules).

We introduce a supplementary transition $(P, \varepsilon) \xrightarrow{b} (Q, \mathfrak{d}')$, which may be (inadvertently) used on other paths than $w_1cw_2rw_3$, thus modifying the LTS in an unacceptable manner. We then fix a brand new state P'' and we inspect all the paths in the LTS other than $w_1cw_2rw_3$ for use of $(P, \delta) \xrightarrow{b} (Q, \mathfrak{e}\delta)$. If such a path is found, we then rename P to P'' on that path (in the preceding transition that will now lead to P'' instead of P and in the current transition which will now start from P'' instead of P). We do the same for the newly introduced rule $P_\perp = r' \to Q$. Then no path other than $w_1cw_2rw_3$ will contain references to P, so no other path will be changed by the introduction of the new transitions.

We perform this procedure for every possible path $w_1cw_2rw_3$ and we obtain a CVP process from which r has been eliminated (that is, hidden). Hiding a return but not its balanced call is similar.

Consider two LTS L' and L'' with initial [final] vPDA states I' and I'' [H' and H'']. We assume without loss of generality that the stack alphabets of L' and L'' are disjoint. The LTS corresponding

to the sequential composition of L' and L'' will have I' as initial vPDA state and H'' as final vPDA state. For all the states $H'_{\delta\perp}$ in L' we make a copy of L'' with identical transitions and with a state $P_{\gamma\delta\perp}$ for every state $P_{\gamma\perp}$ of L''. The copy works the same as the original, but unbalanced returns (introduced by rules of form $P_\perp = r \to Q_\perp$) may now want to match with symbols from δ (which they won't succeed); for every rule $P_\perp = r \to Q_\perp$ we add $\{P_a = r \to Q : a \in \delta\}$ to take care of such a case. We link the copy thus described to $H'_{\delta\perp}$ using a τ transition. Closure under interrupt proceeds with the same construction but we copy L'' for every state $Q_{\delta\perp}$ of L'.

The parallel composition L of L' and L'' will be constructed as follows. The set of vPDA states of L will be the Cartesian product of the vPDA states of L' and L''. The stack alphabet of L will be $\Gamma' \cup \Gamma'' \cup \Gamma' \times \Gamma''$ (with Γ', Γ'' the stack alphabets of L' and L''). The transition relation of L is built as follows:

- Balanced call c and return r are synchronized (similar for synchronized local symbols): For every $P' = c \to Q'_a$ and $P'' = c \to Q''_b$ in L' and L'', respectively, we add $(P', P'') = c \to (Q', Q'')_{(\mathfrak{a},\mathfrak{b})}$. Similarly, for every $P'_a = r \to Q'$ and $P''_b = r \to Q''$ in L' and L'', we add $(P', P'')_{(\mathfrak{a},\mathfrak{b})} = c \to (Q', Q'')$.

- Balanced call c' and return r' in L' and balanced call c'' and return r'' in L'' are unsynchronized (similar for unsynchronized local symbols): For every set of rules $P' = c' \to Q'_a$, $P'' = c'' \to Q''_b$, $R'_a = r' \to S'$, and $R''_b = r'' \to S''$ we add the following rules: $(P', X) = c' \to (Q', X)_a$ for all $X \in \{P'', Q''\}$, $(X, P'') = c'' \to (X, Q'')_b$ for all $X \in \{P', Q'\}$, $(R', X)_\alpha = r' \to (S', X)$ for all $X \in \{R'', S''\}$ and $\alpha \in \{\mathfrak{a}, \mathfrak{b}\}$, and $(X, R'')_a = r'' \to (X, S'')$ for all $X \in \{R', S'\}$ and $\alpha \in \{\mathfrak{a}, \mathfrak{b}\}$.

- Call c' is synchronized, balanced returns r' in L' and r'' in L'' are unsynchronized: We keep c' in L', we rename c' to c'' in L'', and we proceed with c', c'', r', and r'' as in the above case. We then hide c'' (as described above).

- Calls c' in L' and c'' in L'' are unsynchronized, balanced return r' is synchronized: As above, but we rename r' to r'' in L'' and then we hide r''.

It is immediate that the resulting process implements parallel composition. ■

Chapter 5

CVP Trace Semantics

CVP trace semantics produces the set of traces of a CVP process: P_γ is CVP process then $traces(P_\gamma)$ is the set of traces it can produce. $STOP$, $SKIP$, and *RUN* are same in CSP and CVP. In both cases *RUN* can perform any sequence of event any time but the CVP *RUN* operates on a visible alphabet.

5.1 Prefix Choice

An observation of the process $((x : A \rightarrow P(x))_\gamma$ has two possibilities: Either no event has yet occurred, or else an event $a \in A$ has occurred, and the subsequent behavior is that of the corresponding process $P(a)_{\gamma'}$. If $a \in A_l$ then $\gamma' = \gamma$:

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

If $a \in A_c$ then $\gamma' = a\gamma$:

$$\textit{traces}((x : A \rightarrow P(x))_\gamma) = \{\langle\rangle\} \cup \{\langle a\rangle.\textit{tr} | a \in A_c \wedge \textit{tr} \in \textit{traces}(P(a)_{a\gamma})\}$$

If $a \in A_r$ & $\gamma \neq \perp$ then $a\gamma' = \gamma$:

$$\textit{traces}((x : A \rightarrow P(x))_{a\gamma}) = \{\langle\rangle\} \cup \{\langle a\rangle.\textit{tr} | a \in A_r \wedge \textit{tr} \in \textit{traces}(P(a)_\gamma)\}$$

If $a \in A_r$ & $\gamma = \perp$ then $\gamma' = \gamma$:

$$\textit{traces}((x : A \rightarrow P(x))_\perp) = \{\langle\rangle\} \cup \{\langle a\rangle.\textit{tr} | a \in A_r \wedge \textit{tr} \in \textit{traces}(P(a)_\perp)\}$$

$(x : A \rightarrow P(x)_{\gamma'}) = (((x : A \rightarrow P(x))_{\gamma})$, means after execution of the event $x \in A$ the stack will be γ'. So the above four rules can be written together as a single rule::

$$\mathit{traces}(x : A \rightarrow P(x)_{\gamma'}) = \{\langle\rangle\} \cup \{\langle a\rangle.\mathit{tr} | a \in A \wedge \mathit{tr} \in \mathit{traces}(P(a)_{\gamma'})\}$$

Let $P_{aa\perp}$ and $P_{a\perp}$ are two CVP processes with same process definition: $P_a = a \rightarrow P$, $P_\perp = STOP$. $traces(P_{aa\perp}) = \{\langle\rangle, \langle a\rangle, \langle a, a\rangle\}$ and $traces(P_{a\perp}) = \{\langle\rangle, \langle a\rangle\}$. So they are not equivalent processes as $traces(P_{aa\perp}) \neq traces(P_{a\perp})$.

5.2 External Choice

An observer of the choice construct $P_\gamma \Box Q_\delta$ might observe an execution of P_γ or Q_δ; there is no other possibility. The choice operator splits a process in alternative processes so these alternative processes have same stack.

$$\mathit{traces}(P_\gamma \Box Q_\delta) = \mathit{traces}(P_\gamma) \cup \mathit{traces}(Q_\delta)$$

Figure 5.1 presents the laws of external choice. The first three laws are inherited from the properties of the union operator. Law $\Box - unit$ states that external choice gives any process P_γ precedence over $STOP$, which can never resolve a choice in its favor. Law $\Box - zero$ states that external choice allows any process P_γ to be masked by *RUN*: in a choice with *RUN*, if the choice does happen to be resolved in favor of P_γ, then any trace corresponding to such an execution of P_γ is also possible for *RUN*. In algebraic terms, $STOP$ is a unit of external choice, and *RUN* is a zero. Law $\Box - step$ shows that an external choice of two menu choices may be rewritten as a single menu choice.

5.3 Internal Choice

The internal choice $P_\gamma \sqcap Q_\delta$ behaves either as P_γ or as Q_δ, and its environment exercises no control over the decision.

$$\mathit{traces}(P_\gamma \sqcap Q_\delta) = \mathit{traces}(P_\gamma) \cup \mathit{traces}(Q_\delta)$$

Figure 5.2 presents the laws of internal choice. The first three laws are inherited from the union

$$P_\gamma \Box P_\gamma = P_\gamma \qquad \Box - idem$$

$$P_\gamma \Box (Q_\delta \Box R_\beta) = (P_\gamma \Box Q_\delta) \Box R_\beta \qquad \Box - assoc$$

$$P_\gamma \Box Q_\delta = Q_\delta \Box P_\gamma \qquad \Box - sym$$

$$P_\gamma \Box STOP = P_\gamma \qquad \Box - unit$$

$$P_\gamma \Box RUN = RUN \qquad \Box - zero$$

$$(x : A \to P(x))_\gamma \Box (y : B \to Q(y))_\delta = (x : A \to P(x)_{\gamma'}) \Box (y : B \to Q(y)_{\delta'})$$
$$= z : A \cup B \to R(z)_{\gamma' \cup \delta'}$$
where $R(c)_{\gamma' \cup \delta'}$
$$= P(c)_{\gamma'} \quad \text{if} \quad c \in A \setminus B$$
$$= Q(c)_{\delta'} \quad \text{if} \quad c \in B \setminus A$$
$$= P(c)_{\gamma'} \sqcap Q(c)_{\delta'} \quad \text{if} \quad c \in A \cap B \qquad \Box - step$$

Figure 5.1: Laws for external choice

operator. This form of choice has different executions than the external choice $P_\gamma \Box Q_\delta$, since the choice is first resolved by a τ transition before the real choice begins its execution. However, this internal transition is not recorded in any trace, and a trace observer is not concerned with identifying where responsibility lies for particular choices, but only with the possible sequences of events. Under these circumstances, the internal and external choice construct are not distinguished.

$$P_\gamma \sqcap P_\gamma = P_\gamma \qquad \sqcap - idem$$

$$P_\gamma \sqcap (Q_\delta \sqcap R_\beta) = (P_\gamma \sqcap Q_\delta) \sqcap R_\beta \qquad \sqcap - assoc$$

$$P_\gamma \sqcap Q_\delta = Q_\delta \sqcap P_\gamma \qquad \sqcap - sym$$

$$P_\gamma \Box Q_\delta = P_\gamma \sqcap Q_\delta \qquad choice - equiv$$

Figure 5.2: Laws for internal choice

5.4 Parallel Composition

A parallel composition $P_{\gamma A}\|_B Q_\delta$ consists of P_γ performing events in A, and Q_δ performing events in B. The processes P_γ and Q_δ synchronize on events in $A \cap B$, and perform their other events independently. Since P_γ is involved in the performance of all events from A, any execution of the parallel composition projected onto A must be an execution of P_γ. Similiarly, any execution projected onto B must be an execution of Q_δ. The traces of $P_{\gamma A}\|_B Q_\delta$ are those sequences of events which are consistent with P_γ and Q_δ. Only events in A or B, or termination, can be performed, so the set of events in the trace must be contained in $(A \cup B)^\checkmark$.

$$\begin{aligned} \mathit{traces}(P_{\gamma A}\|_B Q_\delta) = \{\mathit{tr} \in \mathit{TRACE} \quad | \quad & \mathit{tr} \upharpoonright A^\checkmark \in \mathit{traces}(P_\gamma) \wedge tr \upharpoonright B^\checkmark \in \mathit{traces}(Q_\delta) \\ & \wedge \sigma(tr) \subseteq (A \cup B)^\checkmark\} \end{aligned}$$

Figure 5.3 presents the laws of alphabetized parallel. Law $\| - idem$ is a form of idempotence: if the interface A provided for P_γ allows all of its possible events $\sigma(P_\gamma) \subseteq A$ then the traces of P_γ are the same as the traces of two copies of P_γ running together. Any execution of P_γ can be performed by both copies of P_γ executing together and synchronizing on every event. The intermediate interfaces in Law $\| - assoc$ depend on the order in which components are composed together, but the resulting process is the same in each case. Law $\| - unit$ provides a unit for the parallel operator: the process $\mathit{RUN}_{(A\cap B)^\checkmark}$, which is always prepared to perform any event in the common interface, and hence places no restriction on P_γ's performance of those events. Law $\| - step$ shows how to reduce a parallel composition of prefix choices to a single prefix choice. The events that are initially possible are those that either side can perform without the co-operation of the other, together with those that both are initially ready to perform. The events that are blocked are those that only one side is ready to perform but where the co-operation of both is required. Laws $\| - term\ 1$ and $\| - term\ 2$ are concerned with termination of a parallel composition. If both components are ready to terminate, then termination occurs.

$$P_{\gamma A}\|_A P_\gamma = P_\gamma \text{ if } \sigma(P_\gamma) \subseteq A \qquad \| - idem$$

$$P_{\gamma A}\|_{B \cup C}(Q_{\delta B}\|_C R_\alpha) = (P_{\gamma A}\|_B Q_\delta)_{A \cup B}\|_C R_\alpha \qquad \| - assoc$$

$$P_{\gamma A}\|_B Q_\delta = Q_{\delta B}\|_A P_\gamma \qquad \| - sym$$

$$\begin{array}{l} C \subseteq A \wedge D \subseteq B \Rightarrow (x : C \to P(x))_{\gamma A}\|_B (y : D \to Q(y))_\delta \\ = (x : C \to P(x)_{\gamma'})_A\|_B (y : D \to Q(y)_{\delta'}) \\ = (z : ((C \setminus B) \cup (D \setminus A) \cup (C \cap D)) \to R(z))_\alpha \\ = z : ((C \setminus B) \cup (D \setminus A) \cup (C \cap D)) \to R(z)_{\alpha'} \\ \text{where } R(c)_{\alpha'} \\ \qquad = P(c)_{\gamma' A}\|_B (y : D \to Q(y)_{\delta'}) \quad \text{if} \quad c \in C \setminus B \\ \qquad = (x : C \to P(x)_{\gamma'})_A\|_B Q(c)_{\delta'} \quad \text{if} \quad c \in D \setminus A \\ \qquad = P(c)_{\gamma' A}\|_B Q(c)_{\delta'} \quad \text{if} \quad c \in C \cap D \end{array} \qquad \| - step$$

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

$$\begin{array}{l} (x : C \to P(x))_{\gamma A}\|_B SKIP = (x : C \to P(x)_{\gamma'})_A\|_B SKIP \\ = x : C \cap (A \setminus B) \to (P(x)_{\gamma' A}\|_B SKIP) \end{array} \qquad \| - term\,2$$

$$P_{\gamma \widetilde{\Sigma}}\|_{\widetilde{\Sigma}} RUN = P_\gamma \qquad \| - unit$$

$$P_{\gamma A}\|_{\widetilde{\Sigma}} STOP = STOP \qquad \| - zero$$

Figure 5.3: Laws for alphabetized parallel

5.5 Hiding

The process $P_\gamma \setminus A$ for $A \subseteq \widetilde{\Sigma}$ has the same execution as P_γ, except that at any point where P_γ performs an external event from A, the process $P_\gamma \setminus A$ performs the same event internally; as a result that event does not appear in the trace.

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

Figure 5.4 shows the laws of Hiding. The first law states that hiding successive sets of events obtains the same process as hiding all the sets of events at once. Second law states that if there is no event then we there is noting to hide. The third and fourth laws are special instances of hiding over a prefix choice. In the first case none of the choice events is hidden, resulting in the same choice of

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

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

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

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

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

Figure 5.4: Laws for hiding

events being offered. In the second case all of the choice events are hidden, resulting in the choice between the subsequent processes. The last law states that hiding does not affect termination.

5.6 Renaming

A forward renamed process $f(P)_\gamma$ behaves the same way as P_γ but performs $f(a)$ whenever P_γ would have performed a. Its traces are the traces of P_γ with every event mapped through f.

$$\mathit{traces}(f(P_\gamma)) = \{f(\mathit{tr}) | \mathit{tr} \in \mathit{traces}(P_\gamma)\}$$

The backward renaming operator $f^{-1}(P_\gamma)$ also behaves in a similar fashion to P_γ, but any event a that is performed by $f^{-1}(P_\gamma)$ corresponds to an event $f(a)$ performed by P_γ. Hence a trace tr of $f^{-1}(P_\gamma)$, when mapped through the function f, must yield a trace $f(tr)$ of P_γ.

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

Figure 5.5 represents the laws of renaming. The first law states that if the mapping f is one-to-one, then renaming with f has a straightforward interaction with prefix choice. A choice of events from C becomes a choice of events from $f(C) = \{f(c) | c \in C\}$. The fact that f is injective means that the event y chosen corresponds to exactly one event $x(= f^{-1}(y))$ from the original choice of

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

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

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

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

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

Figure 5.5: Laws for Renaming

events from C, so the subsequent behavior is that of $P(x)_\gamma$ transformed through f. The second law states that if a process initially is prepared to perform any event from C, then the initial choice for its renamed process is the set of events $f(C)$. However, the result of choosing y could be any of the processes which follow an event mapping to y: if a and b both appear in C, and f maps them both to the same event c, then the renamed process is in effect offering c in two different ways, once resulting from a and once resulting from b. All of the $term$ laws state that the various sorts of renaming cannot affect a process ability to terminate. The interaction between backward renaming and prefix choice is straightforward.

5.7 Sequential Composition

The sequential composition $P_\gamma; Q_\delta$ behaves as P_γ until P_γ terminates successfully, at which point it passes control to Q_δ. Since the termination of P_γ does not denote termination of the entire construct, P_γ's ✓ event is made internal. The traces of $P_\gamma; Q_\delta$ fall into two categories: trace of P_γ before termination, and terminating traces of P_γ followed by traces of Q_δ.

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

There are a number of laws appropriate to sequential composition. These are given in Figure 5.6.

$$P_\gamma; (Q_\delta; R_\beta) = (P_\gamma; Q_\delta); R_\beta \qquad ; -assoc$$

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

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

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

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

Figure 5.6: Laws for sequential composition

Law $; -assoc$ simply states that sequential composition is associative. The $unit$ laws state that $SKIP$ is a left and right unit of sequential composition. Law $; -step$ states that a prefix choice in a sequential composition is equivalent to a prefix choice of sequentially composed processes. Law $; -zero - l$ is a special case of Law $; -step$, in which no events are initially offered—this yields a left zero for sequential composition.

5.8 Interrupt

The process $P_\gamma \triangle Q_\delta$ executes as P_γ, but at any stage before termination it can begin executing as Q_δ. There are therefore two possibilities for any given trace: it is either a trace of P_γ, or else it is a not necessarily terminating trace of P_γ followed by a trace of Q_δ.

$$\mathit{traces}(P_\gamma \triangle Q_\delta) = \mathit{traces}(P_\gamma) \cup \{tr_1.tr_2 | tr_1 \in \mathit{traces}(P_\gamma) \wedge \checkmark \notin \sigma(tr_1) \wedge tr_2 \in \mathit{traces}(Q_\delta)\}$$

Interrupt satisfies a number of laws, given in Figure 5.7, concerning its interaction with choice and with termination. Law $\triangle - assoc$ states that the interrupt operator is associative. Law $\triangle - step$ shows how a prefix choice interrupted by Q_δ unwinds: either it behaves as Q_δ immediately, or else one of the events of the prefix choice occurs, resulting in the subsequent process which may still be interrupted. Law $\triangle - unit - l$ is a special case of $\triangle - step$ in which a process that does nothing may be interrupted by P_δ. Law $\triangle - unit - r$ states that the process $STOP$ is ineffective as an

$$P_\gamma \triangle (Q_\delta \triangle R_\beta) = (P_\gamma \triangle Q_\delta) \triangle R_\beta \qquad \triangle - assoc$$

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

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

$$(x : C \to P(x))_\gamma \triangle Q_d elta = (x : C \to P(x)_{\gamma'}) \triangle Q_\delta$$
$$= Q_\delta \Box (x : C \to (P(x)_{\gamma'} \triangle Q_\delta)) \qquad \triangle - step$$

$$SKIP \triangle P_\gamma = SKIP \Box P_\gamma \qquad \triangle - term$$

Figure 5.7: Laws for interrupt

interrupting process, since there are no events it can perform to interrupt another process. Finally, $\triangle - term$ states that if termination occurs, then the interrupting process is discarded.

5.9 Recursion

In finite-state process algebrae a recursive process creates a loop from one state back to the same state, so it is defined by a relation of form $P = F(P)$. If we follow the same line of thoughts, a CVP recursive process will be defined by the relation $P_\gamma = F(P_\gamma)$. This would however restrict the recursion to regular recursion. In the general case (that includes self-embedding recursion), the relation that defines a CVP recursive process remains $P = F(P)$! Indeed, a recursive process defines a loop from one *vPDA state* back to the same vPDA state; the stack needs not be the same in the two occurrences of the same state, and its behaviour is governed by the processing taking place according to F.

Once this is established we can introduce the stack. We then consider the recursive definition $P = F(P)$ within its proper place (as a CVP process), i.e., $(P = F(P))_\gamma$, or equivalently $P_\gamma = F(P)_\gamma$; γ is the stack content of the process under scrutiny. Note that we are in effect saying that the vPDA state P with stack γ behaves the same as the vPDA state $F(P)$ with the same stack content. We then have that $traces(P_\gamma) = traces(F(P)_\gamma)$. The recursive definition defines an *equation* which must be satisfied by the set $traces(P_\gamma)$. In fact, $traces(P_\gamma)$ is a *fixed point* of the function

on trace sets represented by the CVP expression F; when that function is applied to $trace(P_\gamma)$ to obtain $trace(F(P)_\gamma)$, then the result is again $traces(P_\gamma)$.

Every process contains the empty trace as one of its possible traces, so $\langle\rangle \in traces(P_\gamma)$; that is, $traces(STOP_\gamma) \subseteq traces(P_\gamma) \Rightarrow traces(F(STOP)_\gamma) \subseteq traces(F(P)_\gamma) = traces(P_\gamma)$. This is justified because all of the CVP operators are *monotonic* with respect to $\subseteq$: if $traces(P_\gamma) \subseteq traces(Q_\gamma)$, then $traces(F(P)_\gamma) \subseteq traces(F(Q)_\gamma)$ for any function F constructed out of CVP operators and terms. From standard induction we get for any n $traces(F^n(STOP)_\gamma) \subseteq traces(F(P)_\gamma) = traces(P_\gamma)$ which corresponds 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 correspond to the finite unwindings of the recursive definition, so between them they cover all of the possible traces of $(P = F(P))_\gamma$. Hence

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

.

5.10 Abstract

A process can contain several modules. Abstract extracts only the internal trace of the first module of a process then it follows the rest of the original trace of the process. A complete sub-module is a sub-module which returns to its parent module. Therefore all the complete traces of every complete sub-module are balanced and so no unbalanced call-return can appear in a complete sub-module trace. An incomplete sub-module can appear only at the end of the trace but the incomplete sub-module cannot have any unbalanced return. After the end of the execution of the first module abstract stops working, so the remaining trace of the process remains unchanged.

$$\mathit{traces}(\overline{P_\gamma}) = \{\mathfrak{A}(tr) | \mathit{tr} \in \mathit{traces}(P_\gamma)\}$$

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. $\mathfrak{A}$ is defined at the beginning of Subsection 6.1.1 on page 50.

Chapter 6

CVP Trace Specification and Verification

The CVP trace observer knows the visible partition of the events. As a result, the trace observer can detect when the system (vPDA) performs push and pop. This feature facilitates the definition of four significant functions: abstract function, stack extract, module extract, and completeness. These functions can be used to specify some important properties for software verification: abstract function to specify local properties of a module; stack extract to specify stack limits, access control, and concurrent stack properties; module extract to specify any property specific to a module; and completeness to specify partial and total correctness.

6.1 CVP Trace Functions

6.1.1 Abstract Function

The *abstract function* or $\mathfrak{A}(tr)$ extracts the trace tr' where tr' is the trace of $\overline{P_\gamma}$, and tr is the trace of P_γ. The abstract function is defined as follows: $\mathfrak{A}(tr) = \{l_0.c_1.r_1.l_1.c_2.r_2.l_2...c_k.r_k.l_k.w | tr = l_0.t_1.t_2...t_k.w \wedge l_0 \in \Sigma_l^* \wedge \forall x : 1 \leq x \leq k \wedge t_x \in \{c_x.s_x.r_x.l_x, \langle\rangle\} \wedge (s_x = \langle\rangle \vee \forall s' < s_x : (s' = \langle\rangle \vee |s'|_{\Sigma_c} \geq |s'|_{A_r}) \wedge |s_x|_{\Sigma_c} = |s_x|_{\Sigma_r}) \wedge c_x \in \Sigma_c \wedge r_x \in \Sigma_r \wedge l_x \in \Sigma_l^* \wedge (w = \langle\rangle \vee head(w) \in \Sigma_r^{\checkmark} \vee \forall w' < w : (w' = \langle\rangle \vee |w'|_{\Sigma_c} \geq |w'|_{\Sigma_r}))\}$. The use of abstract function will be illustrated in Subsection 6.2.4.

6.1.2 Stack Extract

In a CVP trace the number of call events is equal to the number of stack symbols pushed onto the stack and the number of balanced return events is equal to the number of stack symbols popped off the stack. We can define a new function *stack extract* or $\mathfrak{S}(tr)$ which will extract the stack γ from the CVP trace tr. $\mathfrak{S}(tr)$ is defined as follows: $\mathfrak{S}(tr) = \{c_{i+j}.c_{i+j-1}...c_{i+2}.c_{i+1}\bot | tr' = tr \setminus \widetilde{\Sigma}_l^{\checkmark} \wedge sq = \mathfrak{R}(tr').\bot \wedge sq = s_{i+j+1}.c_{i+j}.s_{i+j}.c_{i+j-1}...s_{i+2}.c_{i+1}.s_{i+1}.r_i.s_i.r_{i-1}...s_3.r_2.s_2.r_1.s_1.\bot \wedge \forall x : 1 \leq x \leq i \wedge r_x \in \{\widetilde{\Sigma}_r \cup \langle\rangle\} \wedge \forall y : 1 \leq y \leq j \wedge c_{i+y} \in \{\widetilde{\Sigma}_c \cup \langle\rangle\} \wedge \forall z : 1 \leq z \leq i+j+1 \wedge s_z \in S^* \wedge S = \{s | \forall s' < s : (s' = \langle\rangle \vee |s'|_{\widetilde{\Sigma}_r} \geq |s'|_{\widetilde{\Sigma}_c}) \wedge |s|_{\widetilde{\Sigma}_r} = |s|_{\widetilde{\Sigma}_c}\}\}$. The use of the abstract function will be illustrated in Subsections 6.2.1 and 6.2.2.

6.1.3 Module Extract

Module extract or $\mathfrak{M}(tr, a)$ extracts from the trace tr the trace tr' of the first module which starts with the call event a. $\mathfrak{M}(tr)$ extracts the trace tr' of the first module from the trace tr: $\mathfrak{M}(tr) = \{tr' | tr = tr'.tr'' \wedge \forall t < tr' : (t = \langle\rangle \vee |t|_{\widetilde{\Sigma}_c} \geq |t|_{\widetilde{\Sigma}_r}) \wedge (tr'' = \langle\rangle \vee tr'' = \langle\checkmark\rangle \vee (head(tr'') \in \widetilde{\Sigma}_r \wedge |tr'|_{\widetilde{\Sigma}_c} = |tr'|_{\widetilde{\Sigma}_r}))\}$ and $\mathfrak{M}(tr, a) = \{tr' | tr = tr''.a.tr''' \wedge |tr''|_a = 0 \wedge tr' = \mathfrak{M}(tr''')\}$. The use of abstract function will be more clrear in Subsection 6.2.4 and in Subscetion 6.2.5.

6.1.4 Completeness

Completeness or $\mathfrak{C}(tr, a)$ is a function which verifies whether a trace tr contains the complete trace of a sub-module (invoked by call event a) including the call a and its balanced return. If tr contains the desired trace it will return that trace otherwise it will return the empty trace:$\mathfrak{C}(tr, a) = \{tr' | (tr = tr''.tr'.tr''' \wedge head(tr') = a \wedge foot(tr') \in \widetilde{\Sigma}_r \wedge t < tr' : (t = \langle\rangle \vee |t|_{\widetilde{\Sigma}_c} \geq |t|_{\widetilde{\Sigma}_r}) \wedge |tr'|_{\widetilde{\Sigma}_c} = |tr'|_{\widetilde{\Sigma}_r} \wedge |tr''|_a = 0) \vee tr' = \langle\rangle\}$. The use of abstract function will be illustrated in Subsection 6.2.5.

6.2 CVP Trace Specification

CVP behaves as CSP when all the events are locals so CVP can specify any property CSP can. CVP can also specify many important properties for software verification which a regular or a context-free

process algebra cannot specify. The most important among such properties are described below:

6.2.1 Access Control

CVP can specify the access control properties of a module: a module can be invoked if a certain property holds. For example, a procedure A (called by a call event a) can be invoked only if another procedure B (called by a call event b) is in the stack. This property can be expressed by $S(tr) = |\mathfrak{S}(tr)|_b \neq 0 \Rightarrow |\mathfrak{S}(tr)|_a \neq 0$.

6.2.2 Stack Limit

Whenever the stack size is bounded by a given constant, a property holds. For example, if stack size is less than 7, then there will be no no occurrence of an event a in the trace. This predicate is expressed as $S(tr) = |\mathfrak{S}(tr)| < 7 \Rightarrow |tr|_a = 0$.

6.2.3 Concurrent Stack Properties

A concurrent stack property is defined as one stack property holding in a process and another stack property holding in a concurrent process. This property cannot be specified in any context free or regular process algebra. From the trace of $P_{\gamma A}\|_B Q_\delta$ one can specify that if one stack property p holds in P_γ, then another stack property q holds in Q_δ. If tr is the trace of $P_{\gamma A}\|_B Q_\delta$, then $tr \upharpoonright A^\checkmark$ is the trace of P_γ and $tr \upharpoonright A^\checkmark$ is the trace of Q_δ. Indeed, one can use the fact that $\mathfrak{S}(tr \upharpoonright A^\checkmark)$ is the stack of P_γ and $\mathfrak{S}(tr \upharpoonright B^\checkmark)$ is the stack of Q_δ. For example, if stack size is less than 7 in process P_γ, then there will be no invocation for module A (called by a call event a) in Q_δ. This predicate is expressed as $S(tr) = |\mathfrak{S}(tr \upharpoonright A^\checkmark)| < 7 \Rightarrow |\mathfrak{S}(tr \upharpoonright B^\checkmark)|_a = 0$, where tr is the trace of $P_{\gamma A}\|_B Q_\delta$.

6.2.4 Internal Properties of a Module

From a CVP trace one can extract the internal trace of a (recursive) module and then specify any trace properties on that internal trace. Consider a recursive module A (which is called by call event a) in process P_γ. We can then extract the trace of module A using $\mathfrak{M}(tr, a)$, where tr is the trace of P_γ. Using the abstract function we can further extract the internal trace: $\mathfrak{A}(\mathfrak{M}(tr, a))$ is the

internal trace of module A. For instance, that the number of b events is always larger or equal than the number of c events in the local execution of A can be expressed by $S(tr) = |\mathfrak{A}(\mathfrak{M}(tr, a))|_b \geq |\mathfrak{A}(\mathfrak{M}(tr, a))|_c$.

6.2.5 Pre- and Post-Conditions

One can specify pre-/post-conditions of a module. As a result partial and total correctness can be specified in CVP. Partial correctness of a procedure A specifies that if the pre-condition p holds when the procedure is invoked, then if the procedure terminates, the post-condition q is satisfied upon return. Let module A be invoked by a. During the invocation of A, if some event b always precedes another event c, then if A returns then the number of b events will be smaller than the number of c events in module A. The property specified as: $S(tr) = (tr = tr_1.a.tr_2) \wedge (tr_1 \upharpoonright b = \langle\rangle \Rightarrow tr_1 \upharpoonright c = \langle\rangle) \Rightarrow \mathfrak{C}(a.tr2, a) = \langle\rangle \vee (\mathfrak{C}(a.tr2, a) \neq \langle\rangle \wedge |\mathfrak{M}(tr2)|_b < |\mathfrak{M}(tr2)|_c)$. Total correctness of a procedure specifies that if the pre-condition p holds when the procedure is invoked, then the procedure must terminate and the post-condition q must be satisfied upon return. For instance, 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: $S(tr) = (tr = tr_1.a.tr_2) \wedge (tr_1 \upharpoonright b = \langle\rangle \Rightarrow tr_1 \upharpoonright c = \langle\rangle) \Rightarrow \mathfrak{C}(a.tr2, a) = \langle\rangle \wedge (\mathfrak{C}(a.tr2, a) \neq \langle\rangle \wedge |\mathfrak{M}(tr2)|_b < |\mathfrak{M}(tr2)|_c)$. However, total correctness cannot be specified in the trace model because in the trace model if a trace satisfies a property then any prefix of it must also satisfy that property.

6.3 CVP Trace Verification

We can verify many non-regular specification properties in CVP, including the properties stated in Section 6.2. Although the rules of CVP verification look very similar to the rules of CSP, the main difference between CVP and CSP verification rules is that the CVP verification rules are applied on a visible alphabet whereas the CSP verification rules are applied on a local alphabet. The CVP verification rules can handle the visible alphabet due to the availability of the stack.

6.3.1 Prefix Choice

The prefix choice operator contains a number of component processes: It contains a number of component processes, and the first event that is performed can be 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 $P(a)_{\gamma'}$ where $(x : A \to P(x))_\gamma = x : A \to P(x)_{\gamma'}$. The proof rule is:

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

6.3.2 Choice

The choice process $P_\gamma \Box Q_\delta$ or $P_\gamma \sqcap Q_\delta$ behaves either as P_γ or as Q_δ. If $P_\gamma \vdash S(tr)$ and $Q_\delta \vdash T(tr)$, then the choice process $P_\gamma \Box Q_\delta$ or $P_\gamma \sqcap Q_\delta$ satifies the disjunction of these two specifications:

$$\frac{\begin{array}{c} P_\gamma \ \vdash \ S(tr) \\ Q_\delta \ \vdash \ T(tr) \end{array}}{P_\gamma \Box Q_\delta \ \vdash \ S(tr) \vee T(tr)}$$

and

$$\frac{\begin{array}{c} P_\gamma \ \vdash \ S(tr) \\ Q_\delta \ \vdash \ T(tr) \end{array}}{P_\gamma \sqcap Q_\delta \ \vdash \ S(tr) \vee T(tr)}.$$

Let $P_\perp$ and $Q_\perp$ be two CVP processes. A is a module which can perform only event d, invoked by c and returned by d. B is a module which after performing an event b executes its sub-module A. C is another module which first executes its sub-module A then performs an event h. Process $P_\perp$ invokes module B and ends its execution when B returns, while $Q_\perp$ invokes module C and ends its execution when C returns: $P = a \to P1_a$, $P1 = b \to P2$, $P2 = c \to P3_c$, $P3 = d \to P4$, $P4_c = e \to P5$, $P5_a = f \to P6$, $P6 = STOP$, and $Q = g \to Q1_g$, $Q1 = c \to Q2_c$, $Q2 = d \to Q3$, $Q3_c = e \to Q4$, $Q4 = h \to Q5$, $Q5_g = i \to Q6$, $Q6 = STOP$. So $traces(P_\perp) = \{\langle\rangle, \langle a\rangle, \langle a, b\rangle, \langle a, b, c\rangle, \langle a, b, c, d\rangle, \langle a, b, c, d, e\rangle, \langle a, b, c, d, e, f\rangle\}$, $traces(Q_\perp) = \{\langle\rangle, \langle g\rangle, \langle g, c\rangle, \langle g, c, d\rangle, \langle g, c, d, e\rangle, \langle g, c, d, e, h\rangle, \langle g, c, d, e, h, i\rangle\}$, and $traces(P_\perp \Box Q_\perp) = traces(P_\perp \sqcap Q_\perp) = \{\langle\rangle, \langle a\rangle, \langle g\rangle, \langle a, b\rangle, \langle g, c\rangle, \langle a, b, c\rangle, \langle g, c, d\rangle, \langle a, b, c, d\rangle, \langle g, c, d, e\rangle, \langle a, b, c, d, e\rangle, \langle g, c, d, e, h\rangle, \langle a, b, c, d, e, f\rangle, \langle g, c, d, e, h, i\rangle\}$. $P_\perp$ satisfies the following non-regular property:

module A can be invoked only if a is on the stack, while $Q_\perp$ satisfies another non-regular property: module A can be invoked only if g is on the stack:

$$P_\perp \vdash S(tr) = (|\mathfrak{S}(tr)|_a = 0 \Rightarrow |\mathfrak{S}(tr)|_c = 0)$$

$$Q_\perp \vdash T(tr) = (|\mathfrak{S}(tr)|_g = 0 \Rightarrow |\mathfrak{S}(tr)|_c = 0)$$

Then $P_\perp \Box Q_\perp$ or $P_\perp \sqcap Q_\perp$ meets the specification

$$(|\mathfrak{S}(tr)|_a = 0 \Rightarrow |\mathfrak{S}(tr)|_c = 0) \vee (|\mathfrak{S}(tr)|_g = 0 \Rightarrow |\mathfrak{S}(tr)|_c = 0)$$

6.3.3 Parallel Composition

A trace tr of the process $P_{\gamma A}\|_B Q_\delta$ includes a contribution from P_γ and a contribution from Q_δ, contained within the alphabets $A^\checkmark$ and $B^\checkmark$, respectively. Therefore if $P_\gamma \vdash S(tr)$, then $S(tr) \upharpoonright A^\checkmark$ must hold. Similiarly, if $Q_\delta \vdash T(tr)$, then $T(tr) \upharpoonright B^\checkmark$ must hold. Finally, only events in $A^\checkmark$ or $B^\checkmark$ are possible for the parallel com, so it follows that $\sigma(tr) \subseteq (A \cup B)^\checkmark$. This leads to the following proof rule:

$$\frac{\begin{array}{c} P_\gamma \ \vdash \ S(tr) \\ Q_\delta \ \vdash \ T(tr) \end{array}}{P_{\gamma A}\|_B Q_\delta \ \vdash \ S(tr \upharpoonright A^\checkmark) \wedge T(tr \upharpoonright B^\checkmark) \wedge \sigma(tr) \subseteq (A \cup B)^\checkmark}.$$

This rule demonstrates the way in which parallel composition corresponds to conjunction: the constraints S and T both hold, on their respective alphabets.

The parallel composition $P_{\perp\{a,b,c,d,e,f\}}\|_{\{g,c,d,e,h,i\}} Q_\perp$ between the two processes $P_\perp$ and $Q_\perp$ of the Subsection 6.3.2 will satisfy

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

which reduces to

$$|\mathfrak{S}(tr)|_a \wedge |\mathfrak{S}(tr)|_g = 0 \Rightarrow |\mathfrak{S}(tr)|_c = 0 \wedge \sigma(tr) \subseteq \{a,b,c,d,e,f,g,h,i\}^\checkmark$$

6.3.4 Hiding

A trace of the process $P_\gamma \setminus A$ arises from a trace of P_γ simply by removing all the events in A from the trace. Hence for any trace of $P_\gamma \setminus A$ there is a corresponding trace of P_γ. The inference rule thus takes the following form:

$$\frac{P_\gamma \vdash S(\mathit{tr})}{P_\gamma \setminus A \vdash \exists \mathit{tr}_1 : \mathit{tr}_1 \setminus A = \mathit{tr} \wedge S(\mathit{tr}_1)}.$$

The process $P_\perp$ of Subsection 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

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

So for the process $P_\perp \setminus \{c\}$ meets the following specification:

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

6.3.5 Abstract

A trace of the process $\overline{P_\gamma}$ is constructed from the trace of P_γ by removing all the traces of the sub-modules of the top level module. This leads to the following inference rule:

$$\frac{P_\gamma \vdash S(\mathit{tr})}{\overline{P_\gamma} \vdash \exists \mathit{tr}_1 : \mathfrak{A}(\mathit{tr}_1) = \mathit{tr} \wedge S(\mathit{tr}_1)}$$

.

The process $P_\perp$ of Subsection 6.3.2 satisfies the partial correctness property that if b is in the trace when a module is invoked then if the module returns then there will be a d in the trace:

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

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

$$\begin{aligned} S'(tr) = \exists \mathit{tr}_1 : \mathfrak{A}(\mathit{tr}_1) = \mathit{tr} \wedge (\quad & (tr = tr'.a.tr'' \wedge |tr'|_b \neq 0 \wedge a \in \widetilde{\Sigma}_c) \\ & \Rightarrow (\mathfrak{C}(a.tr'', a) = \langle\rangle \vee tr_1.|\mathfrak{M}(tr'')|_d \neq 0) \quad) \end{aligned}$$

6.3.6 Renaming

A trace tr of a renamed process $f(P_\gamma)$ will be a renamed trace $f(tr_1)$ for some tr_1 of P_γ. The inference rule for translating specifications through a forward renaming is then the following:

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

A particular specification S can be translated through f to a specification R. This will be valid provided $R(tr)$ can be shown to translate 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 process $P_\gamma; Q_\delta$ behaves entirely as P_γ until P_γ terminates, after which it behaves as Q_δ. Any given trace of $P_\gamma; Q_\delta$ admits one of the two possibilities: either it is a trace of P_γ which has not yet reached termination, or else it is a trace of P_γ followed by a trace of Q_δ. The proof rule is following:

$$\frac{\begin{array}{c} P_\gamma \vdash S(tr) \\ Q_\delta \vdash T(tr) \end{array}}{P_\gamma; Q_\delta \vdash \neg term(tr) \wedge S(tr) \vee \exists tr_1, tr_2 : tr = tr_1 tr_2 \wedge S(tr_1\langle\checkmark\rangle) \wedge T(tr_2)}$$

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

6.3.8 Interrupt

A trace of the interrupt process $P_\gamma \triangle Q_\delta$ is either a trace of P_γ, or else a non-terminated trace of P_γ followed by a trace of Q_δ. The inference rule is as follows:

$$\frac{\begin{array}{c} P_\gamma \vdash S(tr) \\ Q_\delta \vdash T(tr) \end{array}}{P_\gamma \triangle Q_\delta \vdash S(tr) \vee \exists tr_1, tr_2 : tr = tr_1 tr_2 \wedge \neg term(tr_1) \wedge S(tr_1) \wedge T(tr_2)}.$$

6.3.9 Recursion

If process N_γ is recursively defined by the equation $(N = P)_\gamma$ or equivalently $(N = F(N))_\gamma$, then a rule which is sufficient to establish that $N_\gamma \vdash S(tr)$ is the following:

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

This rule is sound because it provides all the ingredients for establishing by induction that $N_\gamma \vdash S(tr)$. The traces of N_γ are those of $\bigcup_{i\in\mathbb{N}} traces((F^i(STOP))_\gamma)$, all the finite unwindings of $(F(Y))_\gamma$ starting from the process $STOP$. The inductive hypothesis is that $(F^i(STOP))_\gamma \vdash S(tr)$. The side conditions $S(\langle\rangle)$ provides the base case, since it is equivalent to $STOP \vdash S(tr)$, which is same as $(F^0(STOP))_\gamma \vdash S(tr)$.

Chapter 7

Conclusions

We showed that VPL are closed under shuffle and hiding. Together with the already known closure under union, intersection, complementation, renaming, prefix, concatenation, and Kleene star, we showed in effect that VPL have all the necessary closure properties in order for a VPL-based process algebra for infinite-state systems to be possible. We also offered in the process support for the development of the algebra by establishing an LTS semantics for vPDA. Indeed, LTS are the underlying semantic model for all the process algebrae, so this is one significant step.

Finite-state process algebrae have proven useful for the specification and verification of hardware, communication protocols, and drivers. The more complex application software cannot be readily modelled using finite-state mechanisms, as they contain a huge, impractical number of distinct finite states. We therefore believe that an infinite-state process algebra can dramatically open the domain of application software to specification and verification using formal methods (and more specifically algebraic methods such as model-based testing). We thus offered CVP, the first fully compositional visibly pushdown process algebra, as a superset of CSP. The semantics of a vPDA in terms of labelled transition systems establishes the relation between VPL and CVP. Using this vPDA semantics we presented operational and trace semantics for CVP. We proved that CVP is indeed a process algebra, being closed under all its operators.

We then defined some functions on CVP traces which are useful to extract important stack and module information from traces. Using this stack and module information, we showed that unlike

any other existing process algebra one can specify and verify many significant software verification properties in CVP 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. We thus laid the basis of a near future where most of the concurrent process algebraic tools and theories will be based on vPDA instead of finite automata, so that application software will become amenable to formal verification.

7.1 Advantages of CVP over Other Process Algebrae

First, CVP is based on a formalism that goes beyond regular languages and into the context-free realm. For instance, CVP represents recursive modules (or functions), which is not possible in finite-state process algebrae. In addition, CVP can also represent multi-threaded modules; this is possible in the finite state realm, but not in the context-free domain. CVP is at a fortunate crossroad where representing recursive, multi-threaded modules is possible.

One of the major advantages of CVP over context free process algebrae is that by analyzing the trace of a CVP process an observer can determine the content of the stack of that process. This is possible because of the visible nature of CVP; in a CVP trace the call events are in a one-to-one relation with the symbols pushed onto the stack, and the number of balanced return events is equal to the number of stack symbols popped off the stack. Given the one-to-one relation between the set of call events and the set of stack symbols one can reconstruct the current stack from the trace. Stack inspection properties (e.g. a module A should be invoked only within the context of a module B, with no interleaving call to an overriding module C [8]) can thus be specified at trace level. Concurrent processes are possible in CVP, so concurrent stack properties can be specified and verified in CVP.

In CVP, the environment can notice when a module starts (by a call event) and terminates (by a return event). As a result, one is able to specify and verify pre-/post-conditions; pre-conditions of that module can be checked at the starting point, and post-conditions at the end point.

Using the abstract operator the designer can hide the sub-modules from the environment. One can go further and easily create one's own variant of abstract operator, for instance a variant that

only hides a desired sub-module, or hides sub-modules along with their top-level call and return events, or terminate the process just after the end of a module.

By using the module extract function one can easily extract the trace of a module. By using the module extract and abstract functions together one can also extract the local (internal) trace of a module. Unlike context free and regular process algebrae, CVP is thus able to specify and verify the internal properties of a module (e.g., every a should be followed by b in the same module).

7.2 Future

A more concrete operational semantics of CVP (e.g., in terms of vPDA) is appealing. Some groundwork on the matter is presented in our proof of closure, though the process outlined there is not algorithmic (especially as far as the hiding operator is concerned—indeed, there might be infinite paths that need to be considered). We believe that concretizing such a semantics (based on automata) is possible.

This dissertation is only the first milestone on the road to vPDA-based axiomatic verification. We have established a proof system based on the trace model, as the most preliminary proof system for any process algebra. This work opens the whole realm of vPDA-base failures, divergences and infinite (FDI) traces model, pre-order relations, equivalence testing, and so on. The features of CVP presented in Section 7.1 are very powerful so it is our belief that in the near future vPDA-based process algebrae will dominate over context free and regular process algebrae.

Bibliography

[1] R. Alur. Marrying words and trees. In *Proceedings of the 26th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 233 – 242, 2007.

[2] 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.

[3] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM Trans. Program. Lang. Syst.*, 27(4):786–818, 2005.

[4] R. Alur, M. Benedikt, K. Etessami, and M. Yannakakis. Analysis of recursive state machines. In *Proceedings of the 13th International Conference on Computer-Aided Verification*, 2001.

[5] R. Alur, S. Chaudhuri, K. Etessami, and P. Madhusudan. On-the-fly reachability and cycle detection for recursive state machines. *Tools and Algorithms for the Construction and Analysis of Systems*, 3440:61–76, 2005.

[6] R. Alur, S. Chaudhuri, and P. Madhusudan. A fixpoint calculus for local and global program flows. *SIGPLAN Not.*, 41(1):153–165, 2006.

[7] R. Alur, S. Chaudhuri, and P. Madhusudan. Languages of nested trees. In *Proceedings of the 18th International Conference on Computer-Aided Verification*, 2006.

[8] 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.

[9] R. Alur, V. Kumar, P. Madhusudan, and M. Viswanathan. Congruences for visibly pushdown languages. In *Proceedings of the 32nd International Colloquium on Automata, Languages, and Programming*, pages 1102–1114, 2005.

[10] 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.

[11] R. Alur and P. Madhusudan. Adding nesting structure to words. In *Proceedings of the 10th International Conference on Developments in Language Theory*, 2006.

[12] M. Arenas, P. Barcel, and L. Libkin. Regular languages of nested words: Fixed points, automata, and synchronization. *Automata, Languages and Programming*, 4596:888–900, 2007.

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

[14] J. Baran and H. Barringer. A grammatical representation of visibly pushdown languages. *Logic, Language, Information and Computation*, 4576:1–11, 2007.

[15] V. Bárány, C. Löding, and O. Serre. Regularity problems for visibly pushdown languages. In *STACS 2006*, pages 420–431. SpringerVerlag, 2006.

[16] D. Bergamini, N. Descoubes, C. Joubert, and R. Mateescu. Bisimulator: A modular tool for on-the-fly equivalence checking. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2005*, Edinburgh, Scotland, 2005.

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

[18] 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.

[19] L. Bozzelli. Alternating automata and a temporal fixpoint calculus for visibly pushdown languages. *CONCUR 2007 Concurrency Theory*, 4703:476–491, 2007.

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

[21] 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.

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

[23] S. D. Bruda. Preorder relations. *Model-Based Testing of Reactive Systems: Advanced lectures*, 2005.

[24] D. Carotenuto, A. Murano, and A. Peron. 2-visibly pushdown automata. In *Developments in Language Theory*, pages 132–144, 2007.

[25] D. Caucal. Synchronization of pushdown automata. *Developments in Language Theory*, 4036:120–132, 2006.

[26] J. Chabin and P. Réty. Visibly pushdown languages and term rewriting. *Frontiers of Combining Systems*, 4720:252–266, 2007.

[27] S. Chaudhuri and R. Alur. Instrumenting C programs with nested word monitors. *Model Checking Software*, 4595:279–283, 2007.

[28] P. Chervet and I. Walukiewicz. Minimizing variants of visibly pushdown automata. *Mathematical Foundations of Computer Science*, 4708:135–146, 2007.

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

[30] R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1983.

[31] 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.

[32] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. Cadp 2006: A toolbox for the construction and analysis of distributed processes. In *Proceedings of the 19th International Conference on Computer Aided Verification CAV 2007*, Berlin, Germany, 2007.

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

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

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

[36] V. Kumar, P. Madhusudan, and M. Viswanathan. Minimization, learning, and conformance testing of boolean programs. *CONCUR 2006 Concurrency Theory*, 4137:203–217, 2006.

[37] V. Kumar, P. Madhusudan, and M. Viswanathan. Visibly pushdown languages for streaming XML. In *Proceedings of 16th international conference on World Wide Web*, pages 1053–1062, 2007.

[38] C. Löding, C. Lutz, and O. Serre. Propositional dynamic logic with recursive programs. *Journal of Logic and Algebraic Programming*, 73:51–69, 2007.

[39] C. Löding, P. Madhusudan, and O. Serre. Visibly pushdown games. In *FSTTCS*, pages 408–420. Springer, 2004.

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

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

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

[43] A. S. Murawski and I. Walukiewicz. Third-order idealized algol with iteration is decidable. *Foundations of Software Science and Computational Structures*, 3441:202–218, 2005.

[44] D. H. Nguyen and M. Sudholt. Vpa-based aspects: Better support for aop over protocols. In *SEFM '06: Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, pages 167–176, Washington, DC, USA, 2006. IEEE Computer Society.

[45] D. Nowotka and J. Srba. Height-deterministic pushdown automata. In *Proceedings of the 32nd Mathematical Foundations of Computer Science (MFCS)*, 2007.

[46] C. Pitcher. Visibly pushdown expression effects for XML stream processing. In *Proceedings of Programming Language Technologies for XML (PLAN-X)*, January 2005.

[47] J. F. Raskin and Frdric Servais. Visibly pushdown transducers. *Automata, Languages and Programming*, 5126:386–397, 2008.

[48] G. Rosu, F. Chen, and T. Ball. Synthesizing monitors for safety properties: This time with calls and returns. In *RV*, pages 51–68, 2008.

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

[50] 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.

[51] Jiri Srba. Visibly pushdown automata: From language equivalence to simiulation and bisimulation. In *Annual Conference on Computer Science Logic (CSL 06)*, volume 4207 of *Lecture Notes in Computer Science*, pages 89–103. Springer, 2006.

[52] A. Thomo, S. Venkatesh, and Y. Y. Ye. Visibly pushdown transducers for approximate validation of streaming XML. *Foundations of Information and Knowledge Systems*, 4932:219–238, 2008.

[53] S. L. Torre, P. Madhusudan, and G. Parlato. A robust class of context-sensitive languages. In *LICS '07: Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science*, pages 161–170, Washington, DC, USA, 2007. IEEE Computer Society.

[54] S. L. Torre, P. Madhusudan, and G. Parlato. An infinite automaton characterization of double exponential time. In *CSL*, pages 33–48, 2008.

[55] S. L. Torre, M. Napoli, and M. Parente. On the membership problem for visibly pushdown languages. *Automated Technology for Verification and Analysis*, 4218:96–109, 2006.

...